Containment queries on nested sets

Ibrahim, A.

Award date:
2012

Eindhoven University of Technology
Department of Mathematics and Computer Science

Master's thesis

Containment queries on nested sets

September 6, 2012

Author: ing. A. Ibrahim
a.ibrahim@student.tue.nl

Supervisor and tutor: dr. G.H.L. Fletcher
g.h.l.fletcher@tue.nl

Assessment committee: dr. T.G.K. Calders
dr. G.H.L. Fletcher
dr. A. Serebrenik

Abstract

In this thesis we study the problem of containment queries on nested sets of atomic objects. A nested set is a set that contains non-trivial sets as elements, as opposed to a flat set, a set that only contains atomic elements. In general, given two sets of nested sets R and S, the containment join of R in S is defined as:

$$R \bowtie_{\subseteq} S = \{(r, s) \mid r \in R \wedge s \in S \wedge r \subseteq s\}.$$

Our study is motivated by the use of nested data sets in a wide variety of practical applications, e.g., in XML, scientific workflows, business process management, web mash-ups, NoSQL systems, and complex object/nested relations. We introduce two novel algorithms which use an inverted file as a physical representation, for evaluation of subset containment queries on nested sets. We compare both algorithms by implementing them and subjecting them to extensive experiments where we investigate the influence of data set sizes and distribution, i.e., realistic and synthetic data. Analytic and empirical analyses shows that both algorithms have the same general runtime behaviour. Generally, they both show fast response times with querying of synthetic and real data. We also saw a significant decrease in querying time when we cached a subset of the payloads, on the skewed data sets. In general, both solutions have difficulties handling highly skewed data.

Preface

I would like to express my greatest gratitude to George Fletcher who supervised my graduation project. I would also like to thank the committee members, Alexander Serebrenik and Toon Calders, for reviewing my Thesis and giving me good, critical feedback.

Ahmed Ibrahim

Contents

1 Introduction

In this thesis we study the problem of containment queries on nested sets of atomic objects. A nested set is a set that contains non-trivial sets as elements, as opposed to a flat set, a set that merely contains atomic elements.

Given two sets of nested sets R and S, the containment join of R in S is defined as:

$$R \bowtie_{\subseteq} S = \{(r, s) \mid r \in R \wedge s \in S \wedge r \subseteq s\}.$$

Note that $R \bowtie_{\subseteq} S$ is not necessarily the same as $S \bowtie_{\subseteq} R$.

We study here how to compute $R \bowtie_{\subseteq} S$ when R and S are too big to fit in main memory. Our study is motivated by the use of nested sets in a wide variety of practical applications, e.g., in XML, scientific workflows, business process management, web mash-ups, NoSQL systems, and complex object/nested relations. A further motivation for studying containment queries on nested sets is because to our knowledge no research has been done regarding this topic, so the potential for interesting results is very high.

Contributions

Our novel contributions, in this thesis, are the following:

- A definition of containment joins with nested sets.
- Two algorithms, a top-down and a bottom-up, are presented that handle containment queries on nested sets residing on disk.
- A full scale empirical analysis of our algorithms with a variety of synthetic and real data.

From our investigations we draw the following conclusions:

- our solutions for checking containment queries on nested sets are practical, and
- the solutions show promising results on real data sets.

Overview

We start by discussing preliminaries such as flat sets and the algorithms that check containment on flat sets in Chapter 2. In Chapter 3, we extend the notion of flat sets to nested sets and we present data structures and table structures to represent these. Chapter 4 discusses two algorithms for containment queries on nested sets. These algorithms check for containment via a top down and a bottom up approach. Chapter 5 discusses the setup in which we analyze different data sets (synthetic and real). In Chapter 6, we present and discuss empirical results of our experiments. Finally, in Chapter 7 we draw our conclusions and hint at directions for future research.

2 Flat sets

In this chapter we discuss the notions of containment queries on flat sets since they are widely studied in the literature and give an intuition on how to handle nested sets. We give examples of flat sets, then elaborate the different classes of access methods specialized for supporting containment queries on flat sets and summarize algorithms that are associated with those classes. In the summary, we state which class shows the best general performance.

2.1 Containment queries

Containment queries have a purpose whenever we need to examine membership properties such as "is set x contained in set y?". The three basic containment operators are the subset, set equality and the superset. In this thesis we focus primarily on the subset ($\subseteq$) join.

An example of set relations are shown in Table 2.1.

id	values
a	{2,9}
b	{8,18}
c	{1,3}

id	values
A	{2,4,9}
B	{3,8,18}
C	{1,3,4}
D	{3,4,7}

Table 2.1: Two flat set relations R(left) and S(right) are shown. A subset join $R \bowtie_{\subseteq} S = \{(a, A), (b, B), (c, C)\}$.

A practical example [Mam03] of a flat set containment join: "Consider the join of a job-offers relation R with a persons relation S such that $R.required\_skills \subseteq S.skills$, where $R.required\_skills$ stores the required skills for each job and $S.skills$ captures the skills of persons".

2.2 Flat set algorithms

In the literature there are different classes, i.e., signatures, inverted files and hash-based solutions, that support containment queries. These classes are implemented in different algorithms that evaluate containment queries on flat sets.

2.2.1 Signatures

A signature of a set is a hash value over the content of the set [HM97]. It is a bit field of length $b \in \mathbb{N}$. Signatures are used to represent or approximate sets [HM97].

Example: Suppose we have a hash function $h(j) = j \underline{\text{mod}}\ 4$. The signature of set b (Table 2.1) is computed as follows:

$$h(8) = 0$$
$$h(18) = 2$$

The result

$$sig(b) = sig(\{8, 18\}) = 1010$$

i.e., index 0 gets a 1, index 2 gets a 1 and all other indices remain 0.

A property of signatures [HM97] : if $b \subseteq B$, then $sig(b) \subseteq sig(B)$ (i.e., $sig(b)$ is bit-wise contained in $sig(B)$).
In the previous example, if the signature of B (Table 2.1) is computed, it will result in 1011. Indeed, it holds that $b \subseteq B$ implies $1010 \subseteq 1011$.
It may occur that $sig(b) \subseteq sig(B)$, but $\neg(b \subseteq B)$. This is called a false drop [HM97]. Computing the signatures for b and A (Table 2.1) results in $sig(b) \subseteq sig(A)$, but $\neg(b \subseteq A)$.

To conclude, validating with signatures is merely a pretest (efficient because of pruning). A second validation (element comparisons) is still required for eliminating false drops.

The following algorithms use signatures as an underlying structure:

- NL: Nested-Loop ([HM97])
- SHJ: Signature Hash Join ([HM97])
- SSF: Sequential Signature File ([HM03])
- ST: Signature Tree ([HM03])
- ESH: Extendable Signature Hashing ([HM03])
- PSJ: Partitioned Set Join ([Mam03], [MGM03] and [RPNK00])
- SNL: Signature Nested Loop ([Mam03])
- APSJ: Adaptive Pick-and-Sweep Join ([MGM03])

The signature-based algorithms use this validation to significantly reduce the search space in computing flat set containment.

2.2.2 Inverted file

In [Mam03], the inverted file is described as an alternative index for set-valued attributes. For each set element in the domain D, an inverted list, which we define as S_{IF}, is created with the tuple IDs of the sets that contain this element in sorted order. Table 2.2 shows the (partial) inverted file of Table 2.1.

Domain elements	IDs
1	{C}
2	{A}
3	{B,C,D}
...	....
18	{B}

Table 2.2: The inverted file S_{IF} of relation S in Table 2.1. The domain elements are shown in the left column. The right column shows the records associated with the domain elements.

Example: If query $q = \{1, 3\}$ is applied on the inverted file of Table 2.2, the tuple C qualifies since

$$S_{IF}(1) \cap S_{IF}(3) = \{C\}.$$

If $q = \{2, 3\}$, no tuple qualifies, since

$$S_{IF}(2) \cap S_{IF}(3) = \emptyset.$$

The following algorithms use the inverted file as an underlying structure:

- IF: Inverted Files ([HM03])
- BNL: Block Nested-Loop using inverted file ([Mam03])
- IFJ: Inverted File Join ([Mam03])
- OIF: Ordered Inverted File ([TBV+11])

The inverted file based algorithms use this approach to minimize disk accesses, i.e., only retrieve the payloads with respect to the query, while computing set containment joins ([Mam03]).

2.2.3 Hash-based solutions

The final group of algorithms use multiple hash functions to partition and process set containment. These algorithms have been demonstrated in [MGM03] to work well on sets with small cardinalities (e.g., ≤ 10). The algorithms are:

- DCJ: Divide-and-Conquer Set Join ([MGM02])
- ADCJ: Adaptive Divide-and-Conquer Set Join ([MGM03])

2.3 Discussion

Empirical analysis presented in [HM03] shows that the inverted file has the best general performance, especially with set-valued attributes of low cardinality. The hash-based ESH algorithm outperformed the inverted file for set equality queries. Also, the authors generally state that signature-based index structures have difficulties with Zipfian distribution (i.e., skewed distribution) of the frequency of domain elements in the database and some important query cases such as small query sets for subset queries. We elaborate this type of distribution in Section 5.2.

The results presented in [MGM03] show that the PSJ outperforms the APSJ and the ADCJ when the set cardinalities are very small. For larger cardinalities, APSJ shows an increase in performance in comparison to the other algorithms. Figure 2.1 shows the relationship between the flat set algorithms.

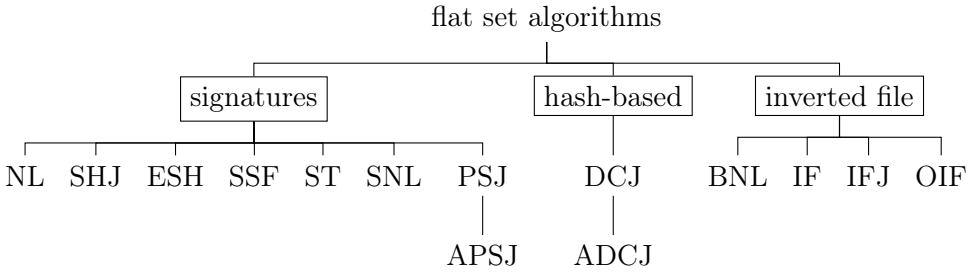


Figure 2.1: The relationship between flat set algorithms. The Figure shows that the APSJ algorithm extends the PSJ algorithm and the ADCJ algorithm extends the DCJ algorithm.

2.4 Summary

The chapter discussed the notion of containment queries on flat sets. We have addressed the classes signatures, inverted file and hash-based solutions. For each of these classes, several different algorithms have been proposed. The results presented in [HM03] show that the inverted file has the best general performance, especially with set-valued attributes of low cardinality. The authors also state that signature-based index structures have difficulties with skewed data and frequently used query sets.

3 Nested sets

In this chapter we extend the notions of containment queries on flat sets to nested sets. To the best of our knowledge, nested set containment queries have not been studied. We start by explaining the difference between flat and nested sets, then we present related work regarding nested sets. Nested sets can also be represented as trees. We briefly discuss different notions of containment on trees since they have different complexities and properties. We summarize the chapter in the last section and discuss why we choose the inverted file, as a physical representation of nested sets.

3.1 Nested sets

In the previous chapter we discussed flat sets. We distinguish flat and nested sets, by allowing nested sets to have non-trivial subsets and flat sets only atomic elements. We need this distinction since the evaluation of containment differs. Table 3.1 shows two nested set relations, where e.g., $a \subseteq A$ and $c \subseteq C$. In general, we want to compute

$$R \bowtie_{\subseteq} S = \{(r, s) \mid r \in R \wedge s \in S \wedge r \subseteq s\}.$$

id	values
a	{2,9, {3,4} }
b	{8,18, {{{4,45}}} }
c	{1,3}

id	values
A	{2,4,9, {3,4, {12,35}}}
B	{3,8,18}
C	{1,3,4, {5,65,34,6,76,87}}
D	{3,4,7}

Table 3.1: An example of two nested set relations R(left) and S(right). A subset join $R \bowtie_{\subseteq} S = \{(a, A), (c, C)\}$.

3.2 Notions of containment

Nested sets can also be represented by trees. Figure 3.1 shows a tree representation of the following two sets:

$$t_r = \{a, b, \{a, b\}, \{b, c\}\} \text{ and } t_s = \{a, b, \{a, b, c\}\}.$$

Figure 3.1: The tree representation of the nested sets relations t_r and t_s. Note that identifiers are used to identify leaf values and children, e.g., nodes(root(t_r))=$\{i, j\}$, leaves(root(t_r))=$\{a, b\}$ and leaves(k)=$\{a, b, c\}$.

For internal node n of tree t, we use the notation nodes(n) to define the non-leaf children nodes of n in t and leaves(n) the leaf values (direct children) under n in t.

Example: Consider the query t_r "is contained" in t_s. There are three natural notions of tree containment:

- Subtree isomorphism, i.e., an injective function maps the nodes in t_r to the nodes of t_s. In the example the query does not hold since t_r has cardinality four and t_s has cardinality three.
- Subtree homomorphism, i.e., the mapping from the nodes in t_r to the nodes of t_s is not necessarily injective. The query does hold because $a \in t_s$, $b \in t_s$, $\{a, b\} \subseteq \{a, b, c\}$ and $\{b, c\} \subseteq \{a, b, c\}$.
- Subtree homeomorphism, i.e., the mapping from the nodes in t_r to the nodes of t_s is not necessarily injective and the query does not necessarily has to match strict descendants (relaxation applies at all levels). Note that the query holds, if we do not consider strict descendants.

The best known main memory, i.e., Random Access Memory, algorithms for subtree homo- and homeomorphism are presented in [KAR12]. The running time of both algorithms is $\mathcal{O}(|r||s|)$, for checking $r \subseteq s$. The algorithms use parallel computing for checking containment. The best known running time for subtree isomorphism is $\mathcal{O}(|r|\sqrt{|r|}|s|/\log(|r|))$ [ST99].

In our work we focus on subtree homomorphism because:

- it is computationally cheaper than isomorphism,
- homomorphism is very natural (e.g., corresponds to standard XPath semantics [GKM09] & [KAR12]), and
- solutions for homomorphism extend easily to the more relaxed homeomorphism and to the stricter isomorphism problem [GKM09] & [KAR12].

We discuss extensions for iso- and homeomorphism in Section 4.6.

3.3 Inverted files for nested sets

In Section 2.2.2 we discussed the inverted file for flat sets. We choose, in our work, to use the inverted file as our physical representation for nested sets because:

- it is widely used in literature, e.g., [HM03] and [Mam03],
- it is especially suitable for real-life applications where the sets are sparse and the domain cardinality large [Mam03] & [HM03], as we saw in Chapter 2,
- it is easier than signatures since we do not have conversions and different representations, and

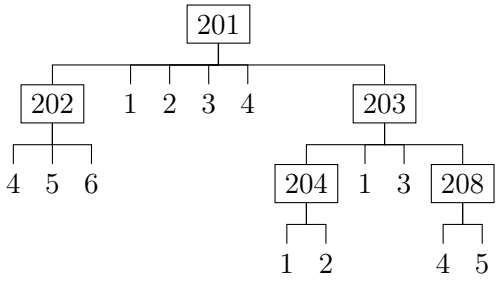
- industrial strength open source solutions for building inverted files are widely available and used, e.g., indexing in search engines.

In order to create the inverted file, it is not sufficient to store the table IDs; we also have to store the non-leaf children, since otherwise containment checks can only happen at the root level. We extend the inverted file for nested sets with the non-leaf children, as shown in Figure 3.2 (right). We can construct a path from the nodes with IDs 201 $\rightarrow$ 208 by matching the children IDs with the node IDs, e.g.,

$$(201,<202,203>) \bowtie (203,<204,208>) = (201,<204,208>)$$

$$(201,<204,208>) \bowtie (204,<>) = (201,<>)$$

The '<>' marker is used to denote a leaf node. Also, the left value (root node ID) never changes since we want to return that value.



leaf	IDs
1	(201, <202,203>), (203, <204,208>), (204,<>)
2	(201, <202,203>), (204,<>)
3	(201, <202,203>), (203, <204,208>)
4	(201, <202,203>), (202,<>), (208,<>)
5	(202,<>), (208,<>)
6	(202,<>)

Figure 3.2: An example of an inverted file that is build on a nested set relation (represented as a tree). The left figure shows the tree and the inverted file is shown on the right.

The inverted file can have multiple representations, i.e.,

- parent-children: $(p, <c_1, ..., c_n>)$, where $c_1 \leq ... \leq c_n$,
- parent-child: $(p, c_1), ..., (p, c_n)$,
- children-parent: $(<c_1, ..., c_n>, p)$, and
- child-parent: $(c_1, p), ..., (c_n, p)$.

We decided to use the first representation, also shown in Figure 3.2 (right), because it simplifies our join algorithms. Further more, it is well-suited for compression, since the intersection only requires the p-values, thus leaving the $c_1, ..., c_n$ compressed; this is a topic for future research. Note that the p-values are also in sorted order, i.e., for every inverted list, $(p_1, <...>), ..., (p_k, <...>)$, in the inverted file, the following holds: $p_1 < ... < p_k$.

3.4 Related work

Sets can be seen as the highest level of abstraction of a hierarchy since they do not maintain any ordering, e.g., the set $\{a, b, \{\{\{\{c\}\}\}\}\}$ has a "nested level" 4, which can be seen as a hierarchical property.

3.4.1 Tree pattern matching

Pattern matching is the process of locating substructures of a larger structure, the target, by comparing them against a given form called the pattern [Kil92]. The problem of nested sets containment given in the previous section can also be viewed as tree pattern matching problems [Kil92].
In [Kil92] the authors state that a node label preserving embedding f of a tree $P = (V, E, root(P))$ in a tree $T = (W, F, root(T))$ is a "path" embedding if it preserves the parent relation. That is, for all nodes u and v: $(u, v) \in E$ if and only if $(f(u), f(v)) \in F$.
Pattern tree P is an unordered path included subtree of target tree T if there is a path embedding of P in T. Clearly, such an embedding exists if and only if the set represented by P is a homomorphic subset of the set represented by T.

Note that our work differs in that we consider bulk containment checks between *sets* of trees. A naive, baseline algorithm we have implemented, for bulk containment checks, is the Nested Loop join, i.e., given relations R and S, check for every $r \in R$ and $s \in S$: $r \subseteq s$. As expected, empirical analysis showed that this approach is not feasible for large data sets.

3.4.2 Xpath

In [HD11] and [GC07] different tree pattern matching techniques are discussed regarding XML query processing. The most popular processing techniques are for XPath [CD99] and XQuery [SCF+07]. An optimized approach of tree pattern matching is twig pattern matching (TPM). TPM is the process of finding in an XML data tree D all matches that satisfy a specified path query pattern Q [GC07]. Twig patterns respect parent-child and ancestor-descendant relationships and express node constraints in formulas with operators such as *equals to* and *contains* [HD11].

Standard TPM solutions use an inverted file representation of XML documents. In our work we are not comparing these standard industrial strength solutions for TPM with our solutions. Our solutions have a different data representation, different encodings and the algorithms use a different approach. It is not a trivial task to understand how to extend XML-join solutions to efficiently handle nested containment queries. We illustrate this by the following example:

Example: Suppose we have the query $q = \{a, b, \{g\}\}$ and our database encoded in XML. In Xpath, the query is formulated as: $/*[/a][/b][/*[/g]]$. The wildcard '*' is defined as "matches any element node". How should we interpret this? Retrieve every list in the inverted file encoded XML database? This would drastically affect the performance with respect to the disk accesses (see also [TBV+11]).

3.4.3 Nested relations

A master student [Boe11] has done research on the nested join operator, for nested relations, in a mediator-based setting. Nested relations can be thought of as a special restricted case of nested sets. In particular, all sets in a nested relation have the exact same structure. Nested relations can be used to describe data on practically all de facto data models such as relational, object-relational and XML-based DBMS. In [Boe11], the author states that information systems companies have difficulties concerning data integration, in their IT-landscape. To solve these difficulties in an efficient way a mediation service, also known as a mediator, is introduced to their IT-landscape. The mediator facilitates communication between different databases and the end-user. The solution presented in that report is a generalized version of the join introduced by [GJ00]. Our work is complementary to this prior work.

3.5 Summary

In this chapter we explained the difference between nested sets and flat sets. Then we discussed nested sets in tree representation and notions of tree containment. There are three basic notions of containment on trees: subtree isomorphism, subtree homomorphism and subtree homeomorphism. We motivated the study of solutions for homomorphic containment. To conclude, we chose the inverted file as a physical access method because of its robustness, suitability for real-life applications and simplicity.

4 Algorithms

In this chapter two algorithms are presented that handle containment queries on nested sets. Section 4.1 discusses Bloom filters that are implemented in the first algorithm. Next, list intersection is discussed since this is an important part of both algorithms. In Section 4.3 both algorithms are demonstrated via an example, followed by the pseudo-code and concluded with the analysis. In Section 4.4 an optimization is presented for both algorithms. Sections 4.5 and 4.6 discuss extensions to both algorithms (e.g., set equality and superset joins). We summarize the chapter in the last Section.

4.1 Bloom filters

In Section 2.2.1 we discuss the signatures that are variously used in flat sets algorithms. Bloom filters are a generalization of signatures that use $k \geq 1$ hash functions, as opposed to signatures that use only one hash function.

4.1.1 Preliminaries

Bloom filters are compact data structures for probabilistic representation of a set that supports membership queries ('is element x in set Y?')[BM02].
In [BM02], Bloom filters are defined for a set $A = \{a_1, ..., a_n\}$ of n elements. The idea is to allocate a vector v of M bits, initially all set to 0, and then choose k independent hash functions, $h_1, .., k_k$, each with range 1 to M. For each element $a \in A$, the bits at position $h_1(a), ..., h_k(a)$ in v are set to 1.

The hash functions $h_1, ..., h_k$ are defined as follows (Universal Hashing [CSRL01]):

$$h_i(a, b, k) = ((ak + b) \underline{\text{mod}}\ P) \underline{\text{mod}}\ M \text{ , where:}$$

- P is large prime (i.e., larger than any key k),
- $a \in \{0, 1, ..., p - 1\}$,
- $b \in \{1, 2, ..., p - 1\}$, and
- M is the number of bits.

Example: Let $P = 492876847$, $M = 9$ and $h_1, ..., h_4$ four hash functions:

- $h_1(k) = ((2k + 5) \underline{\text{mod}}\ P) \underline{\text{mod}}\ M$
- $h_2(k) = ((8556k + 454) \underline{\text{mod}}\ P) \underline{\text{mod}}\ M$
- $h_3(k) = ((345k + 99) \underline{\text{mod}}\ P) \underline{\text{mod}}\ M$
- $h_4(k) = ((45467k + 5346) \underline{\text{mod}}\ P) \underline{\text{mod}}\ M$

If we want to hash the key 123 in the Bloom filter, we need to store $(h_1(123), h_2(123), h_3(123), h_4(123))$. Figure 4.1 shows the result after adding the value 123. Figure 4.2 shows the result after adding keys 123 and 67. Each hash function sets up one bit and it is possible that the same bit is set multiple times (Figure 4.2).

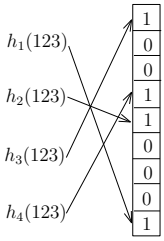


Figure 4.1: The result after adding the key 123 in the Bloom filter. Note that it is possible that a bit can be set multiple times.

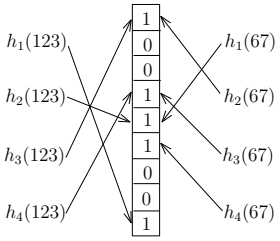


Figure 4.2: The result after adding the keys 123 and 67 in the Bloom filter. This example shows that multiple functions set the same bit.

4.1.2 Hierarchical Bloom filters

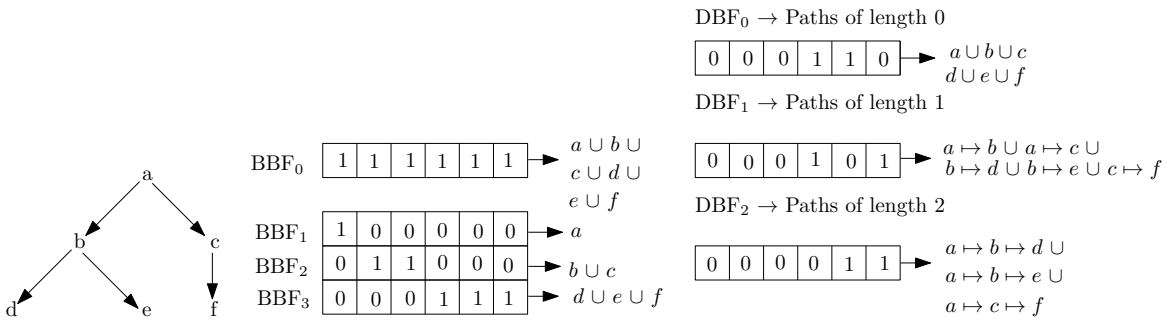
Bloom filters are applicable for flat sets, however they are unable to represent hierarchies [KP04]. In [KP04], two data structures are presented, Breadth and Depth Bloom filters, which are multi-level structures that assist efficient processing of containment queries on XML trees. We do not use these algorithms for our solutions, but we state them to get an idea on how to use Bloom filters in a hierarchy based setting.

Breadth Bloom filter (BFF)

The BBF takes the values at each level of a hierarchical structure and places them in a Bloom filter. A tree of j levels gets $j + 1$ BFFs. For each level $i : 1 \leq i \leq j$ in the tree, we construct a Bloom filter, i.e., BBF_i is the Bloom filter at level i. BBF_0 is a special filter with the logical OR over all other filters. Figure 4.3(b) shows the BBF of Figure 4.3(a). We use a simple hash function that maps a letter to a slot, i.e., a maps to bit 0, b maps to bit 1,..., and f maps to bit 5.

Depth Bloom Filter (DBF)

The DBF takes the values of a path of different length in a tree and places them in a Bloom filter [KP04]. A tree of j levels gets $j - 1$ DBFs. For each level $i : 1 \leq i \leq j - 1$, we construct a Bloom filter, i.e., DBF_i is a Bloom filter at level i. Figure 4.3(c) shows the DBF of Figure 4.3(a). The hash function is the binary number of the path count, e.g., DBF_0 has 6 paths of length 0 and its binary representation is 110.



(a) A simple tree. (b) The BBF of the tree in Figure 4.3(a) (c) The DBF of the tree in Figure 4.3(a)

Figure 4.3: An example of the BBF and DBF data structures. The left most figure shows a simple tree. The middle figure shows the construction of the tree in BBF and the most right figure shows the construction of the tree in DBF.

4.1.3 Bloom filters for containment queries

In the previous Section we discussed hierarchical Bloom filters. These types of filters are well suited for assisting in the evaluation of set containment queries on nested sets. In this Section we show how we can use Bloom filters to assist us in evaluating containment queries. The idea is to prune false matches without investigating the complete tree structure. The Bloom filter contains leaf values taken from a data set. Since it is not feasible to store all leaf values in a Bloom filter, we show how we can find a subset of values to store in the Bloom filter. There are different ways of selecting a subset. One way is to compare Bloom filters by range, as shown in Figure 4.4. Another way is to compare them by level, i.e., the values of the Bloom filters are taken from the same level, as shown in Figure 4.5.

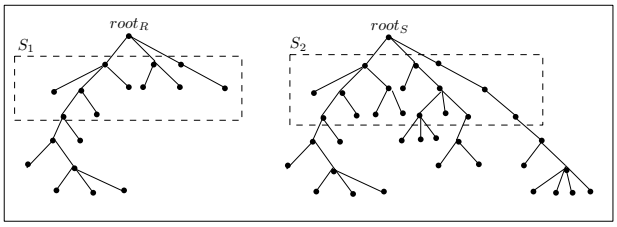


Figure 4.4: An example of two comparable sets by same depth range. Note that a containment check is a latter step. For comparability we are merely interested in the pattern.

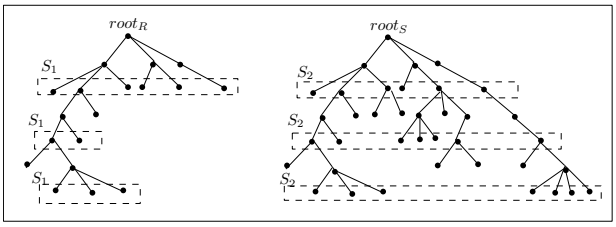


Figure 4.5: An example of two comparable sets that have values taken from the same depth. In this example the values are: depth modulo 2.

4.1.4 Augmenting inverted files with Bloom filters

Figure 4.4 and 4.5 show two different methods of storing leaf values in a Bloom filter. Although these methods take up less space than storing all leaf values, it still may be the case that a large number of leaves are stored in the Bloom filter, e.g., the leaf count at every level is still relatively large. Another method, that we experimented with in our empirical study (Chapter 5), is to store leaf values in the Bloom filter by ranking, e.g., frequency. Suppose we have a Bloom filter that is built on data which has a non-uniform distribution on the leaf values, e.g., some leaf values occur more frequently than other leaf values. Then, it is likely that the Bloom filter contains leaf values that occur frequently in the data. A Bloom filter that contains data with frequently occurring leaf values will prune less data than a Bloom filter that contains the least frequent values.

We extend our inverted file with the Bloom filter:

$$\ell \rightarrow (id, <n_1, ..., n_m>, [v_1, ..., v_k]) \text{ , where:}$$

- ℓ is a leaf value in the data set,
- id is the identifier of the tree,
- $<n_1,...,n_m>$ are the children nodes of id, and
- $[v_1, ..., v_k]$ is an k-bit vector (the Bloom filter) that contains the hash values of the least frequent leaves of the tree.

4.2 List intersection

Recall that in Section 2.2.2 we showed that the inverted file takes the intersection of the set values (Table 2.2):

$$S_{IF}(1) \cap S_{IF}(3) = \{c, C\}$$

In general, for every query node with leaves $\ell_1, ..., \ell_n$, we would like to compute:

$$\bigcap_{1 \leq i \leq n} S_{IF}(\ell_i)$$

4.2.1 Preliminaries

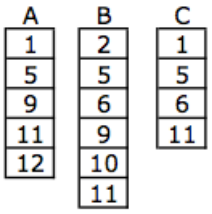
In [CM10], different set intersection algorithms are discussed. The first algorithm iteratively applies the standard two-set intersection algorithm method as a sequence of pairwise operations. The second algorithm combines all the sets and sweeps through them all in a single run. The results presented in that paper show that such a n-way intersection outperforms an iterative 2-way intersection. In Section 4.2.2 we present our adaptation of an n-way list intersection algorithm. We do not compare our list intersection algorithm with the set intersection algorithms presented in [CM10].

4.2.2 Intersection algorithm on inverted files

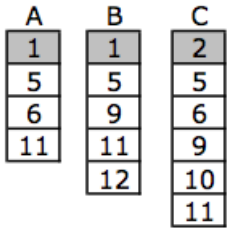
We now present an n-way (n is the number of lists) intersection algorithm for the inverted file. We start with an example, then we give the pseudo code and conclude with the running time analysis.

Example: Let $A =< 1, 5, 9, 11, 12 >$, $B =< 2, 5, 6, 9, 10, 11 >$ and $C =< 1, 5, 6, 11 >$ be sorted lists. Figures 4.6(a)-4.6(i) shows the intuition behind the multi-way intersection algorithm. The algorithm uses the shortest list (of the lists) as a basis, for finding the intersection of all lists.

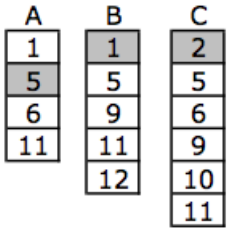
Algorithm 4.1 shows the pseudo code of the intersection algorithm. The first step is sorting the lists by length. Then, we initialize a pointer array to keep track of the current index of each list. We store the intersected results in list R. Lines 6 and 7 are elaborated in Figures 4.6(b)- 4.6(h). These steps involve increasing the pointers. In lines 12 and 13 we state that if a match is found and the last list is reached, we add the result in R. In line 17 we skip to the next value in the smallest list, if every other list, i.e., $T_2, ..., T_n$, has a higher value at their current indices. The process ends when we reach the last index of any list $T_1, ..., T_n$ (line 8).



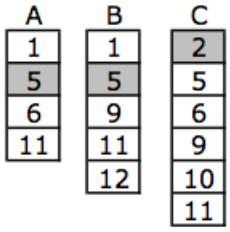
(a) The initial situation (precondition).



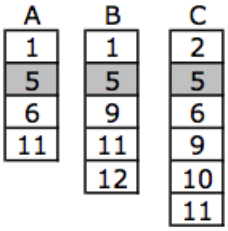
(b) Sort the list of lists by length and initializing a pointer array P that keeps track of the current index of a list. The value of $P[A]<P[C]$, we increase the pointer of A.



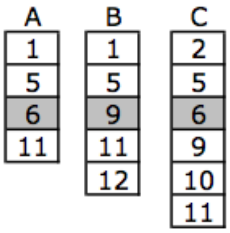
(c) The value of $P[B]<P[A]$, we increase the pointer of B.



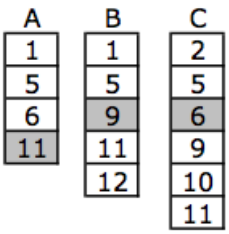
(d) The value of $P[C]<P[B]$, we increase the pointer of C.



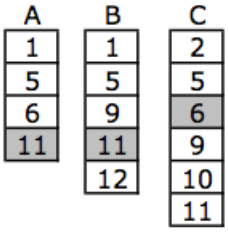
(e) The first result is 5. We output 5 and increase the pointers of all lists.



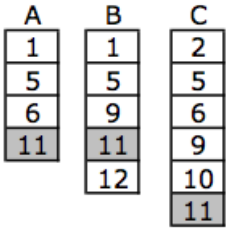
(f) The value of $P[A]<P[B]$, we increase the pointer of A.



(g) The value of $P[B]<P[A]$, we increase the pointer of B.



(h) The value of $P[C]<P[A]$, we increase the pointer of C.



(i) The second result is 11. We stop here, since the end of list A is reached.

Figure 4.6: An example of a multi-way list intersection. The input is a list of three sorted lists. The output is the intersection of these lists, namely $<5, 11>$.

Algorithm 4.1 Multi-way intersection of lists $T_1, ..., T_n$

```
1: Sort T_1, ..., T_n by length
2: Initialize pointer array P = [0,...,0] with length n
3: R ← ∅;
4: for all i: 0 ≤ i < |T_1| do
5:   for all j: 1 < j ≤ n do
6:     while T_j[P[j]] < T_1[i] do
7:       P[j] ← P[j] + 1
8:       if P[j] = |T_j| then
9:         return R
10:      end if
11:    end while
12:    if T_1[i] = T_j[P[j]] then
13:      if j = n then
14:        R ← R ∪ {T_1[i]}
15:      end if
16:    else
17:      skip to next i
18:    end if
19:  end for
20: end for
21: return R
```

Analysis

Every list T_i, for all $1 \leq i \leq n$, is visited at most once. The running time is:

$$\mathcal{O}(\sum_{i=1}^{n} |T_i| + n \log n).$$

If $(n \ll \sum_{i=1}^{n} |T_i|)$, the running time is $\mathcal{O}(\sum_{i=1}^{n} |T_i|)$.

4.3 Containment join algorithms

We next develop two algorithms to compute homomorphic containment queries on collections of nested sets represented as inverted files. The first uses a top-down approach that starts by evaluating the root node and continues with its children nodes. The second algorithm uses a bottom-up approach which starts exploring a tree at its leaf nodes in a Depth-first search fashion. We explain both algorithms via an example, then we give the pseudo code and conclude with the running time analysis.

4.3.1 Top-down approach

The first algorithm we introduce evaluates queries (i.e., containment of nested sets) starting at root of the tree.

Example: Let query $q = \{1, 2, \{1, \{4, 5\}, \{5\}\}\}$ and S_{IF} be the inverted file of Table 4.7.

leaf	IDs
1	(201, <202,203>), (203, <204,208>), (204,<>)
2	(201, <202,203>), (204,<>)
3	(201, <202,203>), (203, <204,208>)
4	(201, <202,203>), (202,<>), (208,<>)
5	(202,<>), (208,<>)
6	(202,<>)

Figure 4.7: An example of an inverted file to illustrate the top-down algorithm (and the bottom-up algorithm we present in the next section).

Figure 4.8(a) shows the tree representation of the query. The top-down algorithm does the following: the algorithm starts at the root of the tree and intersects its leaf values, as shown in Figure 4.8(b). The results (T_1) are passed down to the children of the root. The second step is retrieving the inverted list of the node with leaf 1 ($S_{IF}(1)$ in Figure 4.8(c)) and doing a join that involves matching the children node IDs in T_1 with the IDs in $S_{IF}(1)$ (we explained this process in Section 3.3). The join shown in Figure 4.8(c) is computed as follows:

$$T_2 = T_1 \bowtie S_{IF}(1) \tag{4.1}$$

$$\equiv$$

$$T_2 = ((201,<202,203>),(204,<>)) \bowtie ((201,<202,203>), (203,<204,208>), (204,<>)) \tag{4.2}$$

$$\equiv$$

$$T_2 = (201,<204,208>) \tag{4.3}$$

The same process is repeated for the remaining nodes, as shown in Figures 4.8(d) and 4.8(e). The results of T_3 and T_4 are:

$$T_3 = (201,<>) \text{ and } T_4 = (201,<>) \tag{4.4}$$

The final step is propagating the results up and returning the intersection of the node IDs, i.e.,

$$T_1 \cap T_2 \cap T_3 \cap T_4 = (201) \tag{4.5}$$

Note that the children are not a part of the intersection, e.g., the intersection of

$$T_2 \cap T_4 \tag{4.6}$$

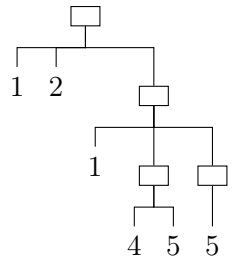
$$\equiv$$

$$(201,<204,208>) \cap (201,<>) \tag{4.7}$$

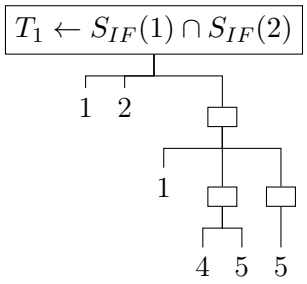
$$\equiv$$

$$(201) \tag{4.8}$$

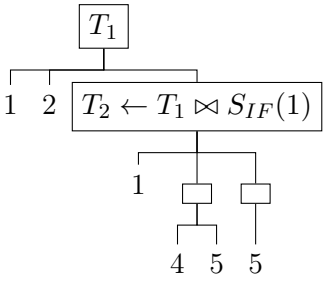
We give the pseudo code of the top down approach in Algorithms 4.2 and 4.3. The algorithm starts with a call to Algorithm 4.2 with a query q and an inverted file encoded database S_{IF}. Note that if the query does not have leaf values, but merely children nodes, we pass the root IDs S_{roots} as a parameter to Algorithm 4.3.



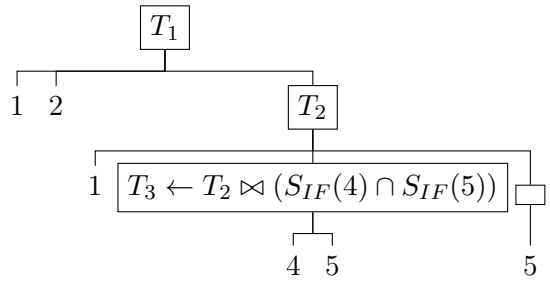
(a) The tree representation of query q



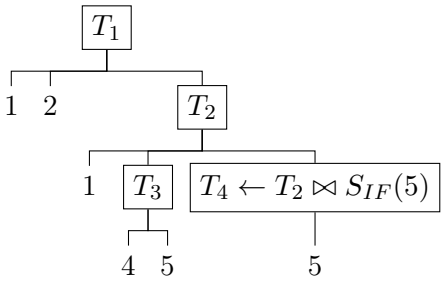
(b) Initially, we evaluate the first node, i.e., the root node.



(c) Next, we evaluate the first child of the root node.



(d) In the third step, we evaluate the left leaf node.



(e) Finally, we evaluate the final leaf node and propagate the information and return the intersected results.

Figure 4.8: An example that shows the execution of the top-down algorithm with the query $q = \{1, 2, \{1, \{4, 5\}, \{5\}\}\}$.

Algorithm 4.2 TD-BNL-Root(query q, S_{IF})

1: **if** leaves(q) $= \emptyset$ **then**
2: $\quad P \leftarrow S_{roots}$
3: **else**
4: $\quad P \leftarrow \bigcap_{\ell \in \text{leaves}(q)} S_{IF}(\ell)$
5: **end if**
6: **return** TD-BNL-HMInterior(nodes(q), S_{IF}, P)

The children nodes are processed in Algorithm 4.3. The variable T_R denotes children nodes of the query. If T_R is empty, we return the results since we cannot continue (in Figures 4.8(d) and 4.8(e), the results are T_3 and T_4). The variable P contains the values that pass the join ($\bowtie$) condition, as shown in Equations 4.1-4.4. Lines 10-13 show the formal definition of the join condition. The variable Q contains the IDs of the retrieved lists that passed the join condition. In line 16, we propagate the results up and returning the intersection, as shown in Equation 4.5.

Algorithm 4.3 TD-BNL-HMInterior(T_R, S_{IF}, P)

```
1:  if T_R = ∅ then
2:     return {p | ∃c : (p, c) ∈ P}
3:  else if P = ∅ then
4:     return ∅
5:  else
6:     Q ← {p | ∃c : (p, c) ∈ P}
7:     for all tuple t ∈ T_R do
8:        P' ← ⋂_{ℓ∈leaves(t)} S_IF(ℓ)
9:        T ← ∅
10:       for all (p1, c1) ∈ P, (p2, c2) ∈ P' do
11:          if p2 ∈ c1 then
12:             T ← T ∪ {(p1, c2)}
13:          end if
14:       end for
15:       Q' ← TD-BNL-HMInterior(nodes(t), S_IF, T)
16:       Q ← Q ∩ Q'
17:    end for
18:    return Q
19: end if
```

Analysis

The worst case running time for a leaf value of q is $\mathcal{O}(|S_{IF}|)$, i.e., the number of trees in the data set. The worst case running time of line 2-4 (Algorithm 4.2) becomes $\mathcal{O}(|\text{leaves}(q)| * |S_{IF}|)$.
The running time of Algorithm 4.3 is:

- line 8. Each leaf node in S_{IF} is called exactly once, worst case is that the complete S_{IF} is fetched: $\mathcal{O}(|\text{leaves}(q)| * |S_{IF}|)$.
- lines 10-14. Worst case is that P contain all trees in data set S. Since both lists are sorted, a merge join is sufficient: $\mathcal{O}(|S|)$.
- line 16. An intersection of two sorted lists having at most $\mathcal{O}(|S|)$.
- line 15. For each node, lines 10-14 and 16 are executed: $\mathcal{O}(|\text{nodes}(q)| * 2|S|)$. Leaving out the constant, the running time becomes $\mathcal{O}(|\text{nodes}(q)| * |S|)$.

The total running time is

$$\mathcal{O}((|\text{leaves}(q)| * |S_{IF}|) + (|\text{nodes}(q)| * |S|)).$$

4.3.2 Bottom-up approach

The second algorithm we introduce evaluates containment in a depth first-search manner. The algorithm uses a stack to store temporary and final results.

Example: We use the same example to explain the intuition behind the algorithm. The algorithm starts by initializing an empty stack at the root of the tree. Figure 4.9(a) shows that a marker '$' is pushed on the stack. Figure 4.9(b) and 4.9(c) show that two markers are pushed onto the stack since two nodes are visited. If the current node has no children, the algorithm intersects the leaf values of the current node, as shown in Figure 4.9(d). The algorithm pops the marker and pushes the head values onto the stack, i.e.,

$$\text{head}(S_{IF}(4) \cap S_{IF}(5)) \tag{4.9}$$

$$\equiv$$

$$\text{head}((202, <>), (208, <>)) \tag{4.10}$$

$$\equiv$$

$$(202, 208) \tag{4.11}$$

The algorithm continues with the next node, as shown in Figure 4.9(e). The current node does not have any children and the inverted list associated with the leaf value 5 is retrieved and pushed onto the stack, as shown in Figure 4.9(f). The algorithm propagates up, retrieves all elements in the stack up to the first encountered marker and pushes back the merged results, i.e., the children of the nodes is $S_{IF}(1)$ are matched with the retrieved elements from the stack, as shown in Figure 4.9(g). The algorithm continues propagating up to the root and repeating the process with leaf values 1 and 2, as shown in Figure 4.9(h). The final step is to pop the stack and return the results.

We give the pseudo code of the bottom-up approach in Algorithms 4.4 and 4.5. The algorithm starts with a call to Algorithm 4.4 with a query q and an inverted file encoded database S_{IF}.

Algorithm 4.4 BU-BNL-Root(query q, S_{IF})

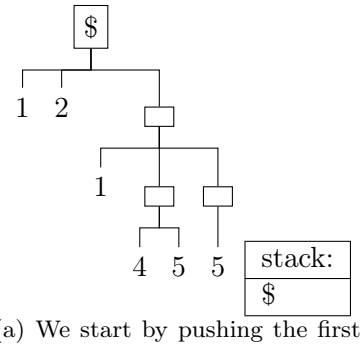
```
1: Stack s ← ∅
2: BU-BNL-HMInterior(root(q), s, S_IF)
3: return pop(s)
```

Lines 1-3, in Algorithm 4.5, correspond to Figures 4.9(a)-4.9(c) and 4.9(e). The formal definition of the matching condition, shown in Figures 4.9(g) and 4.9(h), is presented in Algorithm 4.5 (line 12).

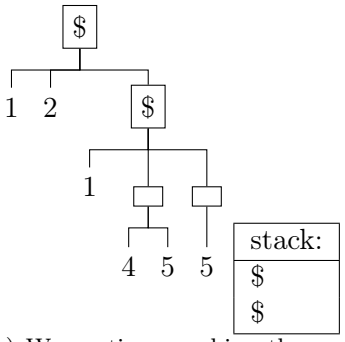
Analysis

The running time of Algorithm 4.5 is :

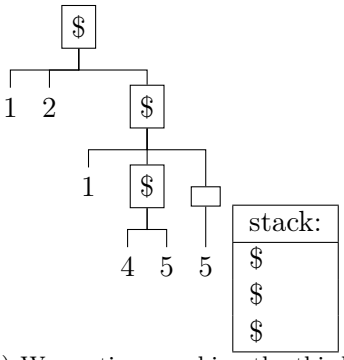
- line 11. In the worst case the complete inverted file is retrieved: $\mathcal{O}(|\text{leaves}(q)| * |S_{IF}|)$.
- line 12. In the worst case both of the sets contain all elements in S. Since both sets are sorted, a merge join is sufficient : $\mathcal{O}(|S|)$.



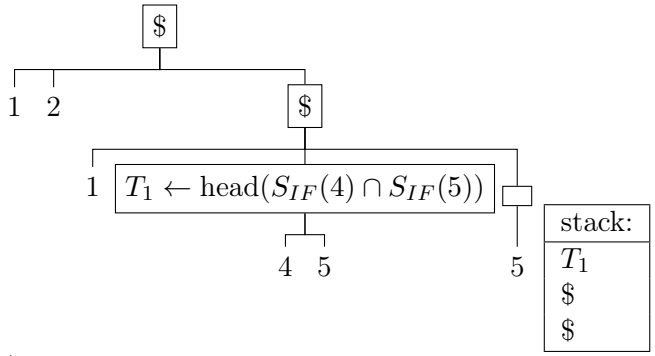
(a) We start by pushing the first marker onto the stack.



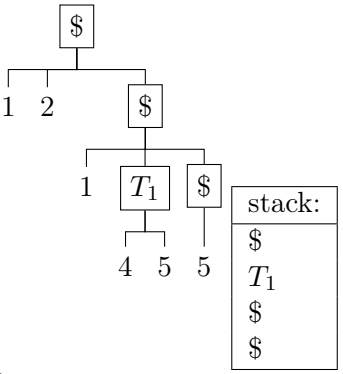
(b) We continue pushing the second marker onto the stack.



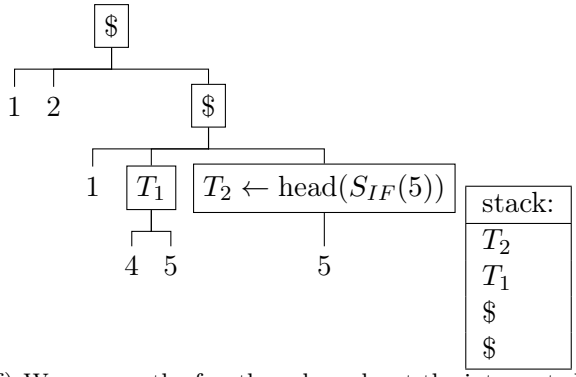
(c) We continue pushing the third marker onto the stack.



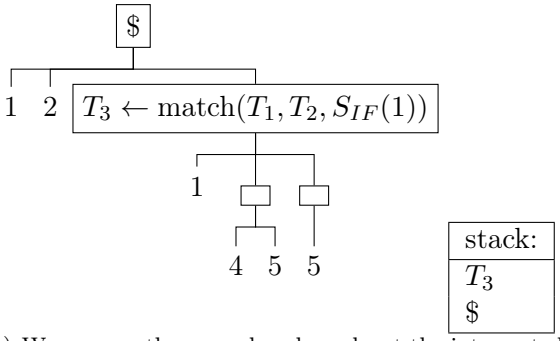
(d) We process the third node and put the intersected values onto the stack.



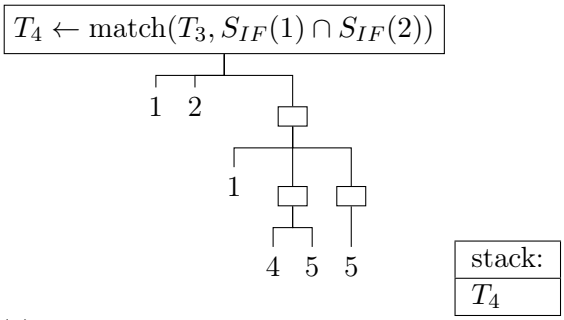
(e) We push the fourth marker onto the stack.



(f) We process the fourth node and put the intersected values onto the stack.



(g) We process the second node and put the intersected values onto the stack.



(h) We process the first node and return the results.

Figure 4.9: An example that shows the execution of the bottom-up algorithm with the query $q = \{1, 2, \{1, \{4, 5\}, \{5\}\}\}$. Note that the bottom-up algorithm uses a data structure (the stack) to store temporary results.

Algorithm 4.5 BU-BNL-HMInterior(Node u, Stack s, S_{IF})

1: push(\$, s)
2: **for all** $c \in$ nodes(u) **do**
3: BU-BNL-HMInterior(c, s, S_{IF})
4: **end for**
5: $Lists \leftarrow \emptyset$
6: **while** peek(s) $\neq$ \$ **do**
7: $Lists \leftarrow Lists \cup \{\text{pop}(s)\}$
8: **end while**
9: pop(s)
10: **if** $\{\forall L \in Lists : L \neq \emptyset\}$ **then**
11: $Pairs \leftarrow \bigcap_{\ell \in \text{leaves}(u)} S_{IF}(\ell)$
12: $Heads \leftarrow \{h \mid \forall L \in Lists : \exists c_i : 1 \leq i \leq n : (h, \{c_1, ..., c_n\}) \in Pairs \wedge c_i \in L\}$
13: push($Heads$, s)
14: **else**
15: push($\emptyset$, s)
16: **end if**

The running time of lines 1-10 are neglectable and will be left out in the total running time. For each child node in q, line 12 is performed. For each leaf value in q, we scan - in the worst case- all of S_{IF}. The total running time is:

$$\mathcal{O}((|\text{nodes}(q)| * |S|) + (|\text{leaves}(q)| * |S_{IF}|))$$

Note that the worst-case running time of the top-down and bottom-up algorithm are equal.

4.4 Caching

The inverted file S_{IF} is accessed continuously in the top-down and the bottom-up approach, i.e., for every leaf ℓ in a query, retrieve $S_{IF}(\ell)$. Retrieving the inverted list for every leaf value is fine for small queries, but for large queries and/or a batch of queries, e.g., a number of queries executed after each other, accessing S_{IF} can incur many random accesses. We can reduce access cost by caching a subset of the leaf values in main memory. In [TPVS06] an inverted file, with an additional trie data structure, is proposed that can efficiently answer queries with set-valued attributes by indexing merely a subset of the most frequent of the items that occur in the indexed relation. The indexing is done with respect to the data set. Table 4.1 shows the top two most frequent item list of the data set with respect to Figure 3.2 (left). Suppose we want to find a match for the set $\{1, 2, \{1, 4, 3, 64\}\}$. The inverted file is accessed three times (instead of six times) since leaf values 2, 3 and 64 are not in main memory.

leaf	frequency
1	3
4	3

Table 4.1: The top two most frequent items of the data set shown in Figure 3.2 (left).

Table 4.1 shows an example of caching with respect to the data set. Another natural notion of caching is with respect to the queries, i.e., based on an observed workload. We leave the study of query workload driven caching open for future research.

4.5 Extension 1: evaluation of other join predicates

Recall the practical example, of a set containment join, we gave in Section 2.1: "Consider the join of a job-offers relation R with a persons relation S such that $R.required\_skills \subseteq S.skills$, where $R.required\_skills$ stores the required skills for each job and $S.skills$ captures the skills of persons". In some cases we want that the

- required skills of a job exactly matches the skills of a person ($R.required\_skills = S.skills$), i.e., set equality join,
- that the person is not over qualified ($R.required\_skills \supseteq S.skills$), i.e., superset join, and
- that the person and job share at least ϵ common requirements, ($|R.required\_skills \cap S.skills| \geq \epsilon$), i.e., ϵ-overlap join.

The top-down and bottom-up algorithms presented in Section 4.3 can also be adapted to process these set-based joins. Note that these adjustments must be implemented in the intersection operation of both algorithms (i.e., line 8 of Algorithm 4.3 and line 11 of Algorithm 4.5).

Let S be a data set and tree $T_1, ..., T_m \in S$. Let S_{IF} be the inverted file of S.

4.5.1 Set equality join

For subset joins that satisfies a query $q : 1 \leq i \leq m : q \subseteq T_i$, the following holds: $|\text{leaves}(q)| \leq |\text{leaves}(T_i)|$. For the set equality join, which is defined as $q = T_i$, the condition is strengthened in both algorithms, for each T_i found in the inverted list, i.e.,

$$S' \leftarrow \bigcap_{\ell \in \text{leaves}(q)} S_{IF}(\ell) \tag{4.12}$$

$$S' \leftarrow \text{remove } T_i \in S' \text{ not satisfying: } |\text{leaves}(q)| = |\text{leaves}(T_i)|. \tag{4.13}$$

4.5.2 Superset join

For subset joins the inverted lists for all elements in q are fetched and intersected. For the superset join, which is defined as $q \supseteq T_i$, a more relaxed nature of processing is required, i.e.,

$$S' \leftarrow \biguplus_{\ell \in \text{leaves}(q)} S_{IF}(\ell) \tag{4.14}$$

$$S' \leftarrow \text{remove } T_i \in S' \text{ not satisfying: } |\text{leaves}(T_i)| = \text{ number of occurrences of } T_i \text{ in } S', \tag{4.15}$$

where $\biguplus$ is the multi-set union. Note that multi-set semantics is required to allow duplicates.

4.5.3 ϵ-overlap join

The superset join retrieves all trees that have at least one common element. For the ϵ-overlap join [Mam03], which is defined as $|q \cap T_i| \geq \epsilon$, every $T_i \in S'$ should appear at least ϵ times, for some fixed $\epsilon \in \mathbb{N}$, i.e.,

$$S' \leftarrow \biguplus_{\ell \in \text{leaves}(q)} S_{IF}(\ell) \tag{4.16}$$

Figure 4.10: An example of an iso- and homomorphic containment, i.e., the left tree, query q, is contained in the right tree s.

$$S' \leftarrow \text{remove } T_i \in S' \text{ that do not appear at least } \epsilon \text{ times.} \tag{4.17}$$

4.6 Extension 2: other embeddings

The algorithms presented in Section 4.3 can also be adapted to evaluate iso- and homeomorphic containment. We illustrate this on the top-down approach.

4.6.1 Isomorphic containment

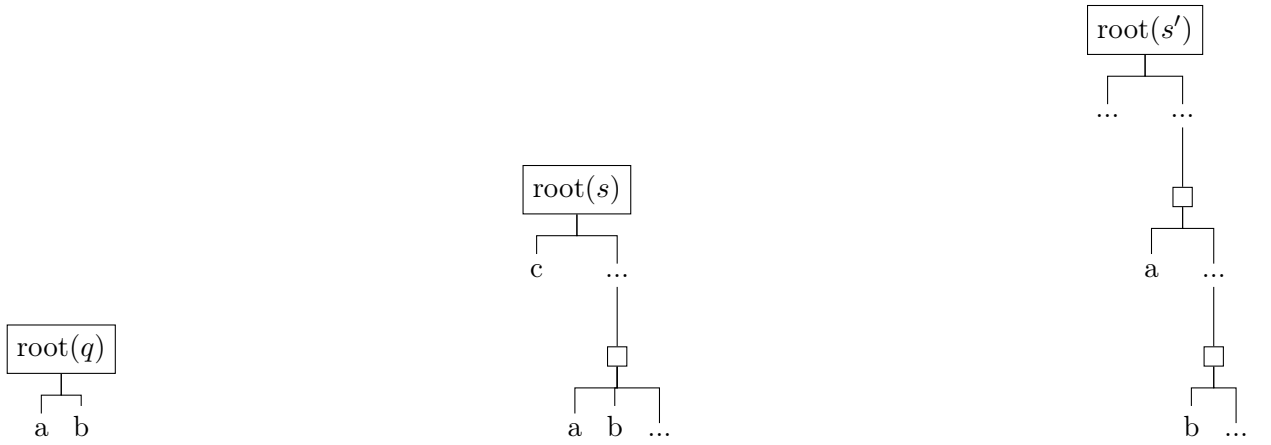
An extra condition that is added in the algorithm: at every level of a query, the matches must be injective. As a result the running time of the algorithm increases significantly, i.e., higher order in running time compared to the running times of the algorithms presented in Section 4.3, since we have to do some backtracking in some cases to find the right containment. Figure 4.10 shows that query q is isomorphic contained in tree s, but node i is not satisfied by node j. Backtracking is required to ensure that $i \rightarrow m$ and $k \rightarrow j$. For homomorphic containment, $i \rightarrow m$ and $k \rightarrow j$ or $k \rightarrow m$.
Isomorphism significantly increases the complexity of the algorithm since we have to do more bookkeeping to ensure that the nodes of the query and the nodes of the data set match. First, we have to mark retrieved lists of the inverted file to ensure that they are not reused again (injective part). Then, we have to check multiple scenarios to ensure that the correct match is made. This means that we also have to unmark marked lists in order to reuse them for new scenarios.

4.6.2 Homeomorphic containment

Figure 4.11(c) shows a type of containment which is not regarded as homeomorphism with respect to the query shown in Figure 4.11(a). In particular, homeomorphism only applies to set nesting and not set membership. Hence, a and b must appear together as leaves.

The algorithm can be changed in two different ways to check for such homeomorphic containment:

Variant 1 First, all nodes in the inverted lists are tagged with a scheme that stores ancestor-descendant relationships (e.g., pre-post ordering [GC07]). Next, the join condition in the algorithm is updated to check ancestor-descendant containment (i.e., lines 10-13 of Algorithm 4.3). This adjustment does not increase the complexity since we can determine the ancestor-descendant relationship between any two nodes in constant time by using only two number-comparison operations [GC07].



(a) query q

(b) q is homeomorphic contained in s

(c) q is not homeomorphic contained in s'

Figure 4.11: The left tree, query q, is homeomorphic contained in the middle tree s but not in the right tree s'. Note that iso- and homomorphic containment is not possible since both s and s' do not satisfy q at the root level.

Variant 2 The $\bigcap$-operator in the algorithm is relaxed such that failures (intersection may return $\emptyset$) are allowed to occur during the evaluation of a query, as shown in Figure 4.11(b). The Figure shows containment is found at a sublevel and not at the root level. This adjustment increases the complexity of the algorithm slightly since we have to add extra recursive calls to allow the relaxation. A direct consequence of adding extra recursion is that the performance of the algorithm deteriorates.

4.7 Summary

In this chapter we introduced our fundamental tools. Bloom filters are similar to signatures, but a major difference is that Bloom filters use $k \geq 1$ hash functions, as opposed to signatures that use a single hash function. Next, an algorithm for list intersection was presented. For both algorithms, top-down and bottom-up, examples were given followed by the pseudo-code and concluded with the analysis of both algorithms. Analysis shows that both algorithms have the same worst-case running time. We also discussed an optimization, namely caching the most frequent leaf values occurring in a data set. An advantage of caching these leaves is that they have the largest payloads and placing them in main memory we hope to obtain less disk IOs. Finally, the chapter is concluded with extensions for iso- and homeomorphism, and evaluation of other join predicates such as the set equality, superset and ϵ-overlap join.

5 Experiment set-up

In this chapter we discuss the experimental set-up to evaluate the top-down and bottom-up algorithms, and the impact of the Bloom filter and caching. The system parameters are described in Section 5.1. Section 5.2 describes the synthetic data used for the experiments. There are two types of synthetic data: uniform and skewed. Then, in Section 5.3 we describe the realistic data and the evaluation of queries (Section 5.4). In Section 5.5 we describe a high level overview on how to construct the inverted file. We conclude the chapter with a summary.

5.1 System parameters

The experiments were conducted on a system that is composed of 7 servers with each 144 GB that are aggregated by the vSMP software to appear as one system. The specifications are Fedora 12 / 64-bit, 56 x 2 Ghz, 935 GB memory (aggregated) and an 2.8 TB local disk. The data structures and algorithms are implemented in Java (version J2SE-1.5), as a single threaded process.

Database The inverted file is implemented in Java, using Tokyo Cabinet[1] as the storage engine. Tokyo Cabinet supports relations with a key and value pair. According to the authors, Tokyo Cabinet replaces conventional DBM products:

- improves space efficiency : smaller size of database file,
- improves time efficiency : faster processing speed,
- improves parallelism : higher performance in multi-thread environment,
- improves usability : simplified API, and
- improves robustness : database file is not corrupted even under catastrophic situation.

We also experimented with other solutions such as, Berkeley DB[2] and cdb[3], but Tokyo Cabinet outperformed those solutions regarding the creation and retrieving of trees in synthetic data, that we used for our experiments (Section 5.2).

[1] http://fallabs.com/tokyocabinet/ (version 1.24)
[2] http://www.oracle.com/technetwork/database/berkeleydb/overview/index-093405.html
[3] http://cr.yp.to/cdb.html

parameter	min value	max value
data set cardinality	125,000	4,000,000
leaf value cardinality	0	10,000,000

Table 5.1: The parameters for generating the data sets. The data sets increase by a factor 2 (i.e., 125k, 250k,...,4000k).

Caching Each benchmark was run with and without caching, as described in Section 4.4. The caching parameter, i.e., the number of lists that are stored in main memory is set to 250.

Assumptions The payloads, i.e., the retrieved inverted lists, fit in main memory. As each intersection is on flat sets, the BNL algorithm of [Mam03] could be used if the payloads do not fit in main memory. The stack used in the bottom-up algorithm always fits in main memory. I/O efficient solutions for stacks, presented in [DKS08], can be used if the payloads do not fit in the main memory.

5.2 Synthetic data

Two different synthetic data sets were used in our experiments. One data set has a uniform distribution of leaf values, another has a Zipfian distribution [4]. For a summary, see Table 5.1.

We devised a function, called RandomTree, that randomly generates and builds a tree T (note that T is a class in Java, Appendix A). The function requires four parameters:

$$\text{RandomTree}(l, c, p, z) : T,$$

where l is the maximal number of leaves at every level of the tree T, c is the maximum number of node children at every level of the tree T, p is the stopping probability ($p \in [0, 1)$) and $z \in \mathbb{R}$ is the Zipfian. This distribution is relevant since a part of our real data (Section 5.3) is also highly skewed. We want to simulate the effects on our synthetic data.
We build a tree by generating leaves and node children per level. The stopping probability p determines the depth of T and eventually stops the building process, i.e, at every level of T, we check:

if Math.random()>p **then** create new node children for T **else return** T

Table 5.2 shows the different tree types used in our experiments.

Example: Suppose we want to create a data sets that contains 500,000 wide trees, with skewness 0.5. We do this by calling function RandomTree(12, 6, 0.8, 0.5) 500,000 times (in a for-loop) and saving the results in a file.

5.2.1 Constructing the inverted file

The top-down and bottom-up algorithm of Section 4.3 share common attributes (identifier, # of child nodes and whether the current node is a root or not) but also have their own attributes. For example, the top-down algorithm uses a Bloom filter (Section 5.4.2) which uses 7 bytes per tree. This additional overhead is undesirable for the bottom-up algorithm since it does not use Bloom filters. To overcome this, two inverted files are implemented for fair comparisons.

[4] Various naturally occurring phenomena exhibit a certain regularity that can be described by Zipf's law, e.g., word usage or population distribution [HM03]. For further discussion see: http://en.wikipedia.org/wiki/Zipf's_law

parameter	wide trees	deep trees
max # of leaves	12	2
max # of child nodes	6	3
stopping probability	0.8	0.2
skewness	0.5, 0.7, 0.9	0.5, 0.7, 0.9

Table 5.2: The parameters for generating different types of trees. Analyses shows that using higher skewness values significantly increases the construction time of the inverted file and lower values tends toward a uniform distribution (a skewness value 0 is uniform).

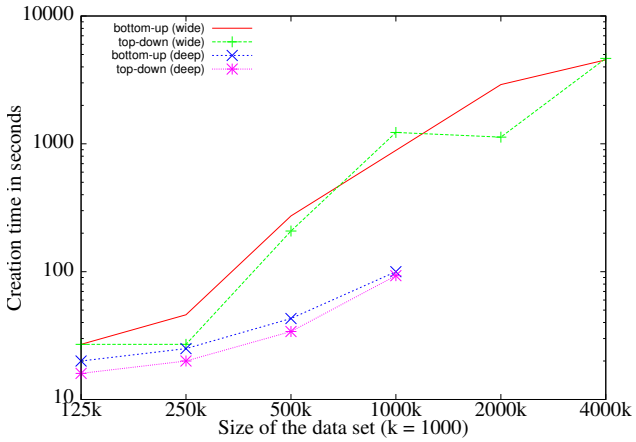


Figure 5.1: The creation time of the inverted files for the uniform data set. The top-down data set (deep) takes the least time to construct. The bottom-up data set (wide) takes the most time to construct.

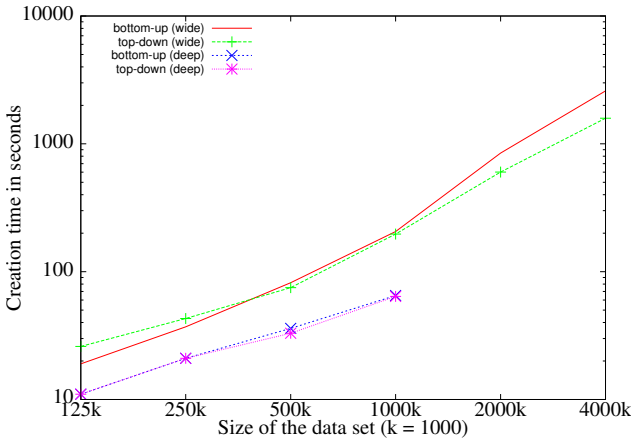


Figure 5.2: The creation time of the inverted files with skewed data. The skewness is set to 0.5. The figure shows an exponential relationship between the different sizes.

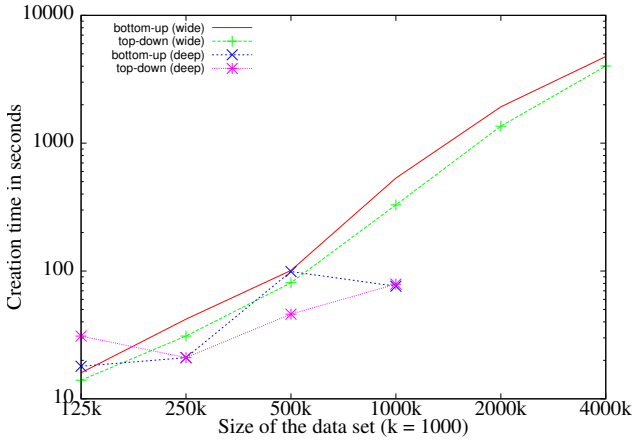


Figure 5.3: The creation time of the inverted files with skewed data. The skewness is set to 0.7. The figure shows an exponential relationship between the different sizes, for the wide data sets. Also, it is not always the case that deep trees are constructed faster than wide trees.

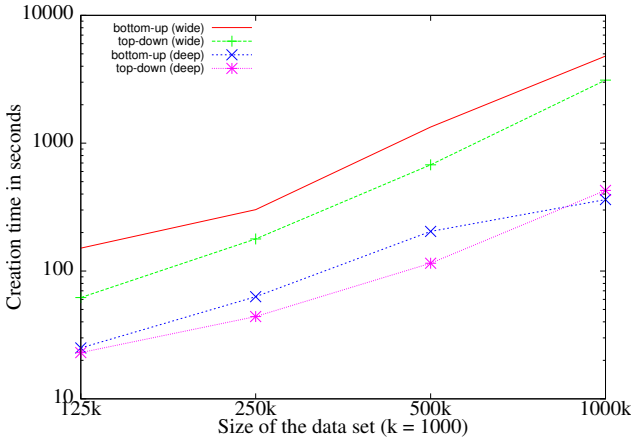


Figure 5.4: The creation time of the inverted files with skewed data. The skewness is set to 0.9. The figure shows an exponential relationship between the different sizes. The figure also shows that wide data sets are constructed faster than deep data sets.

	uniform				Zipf(0.5)			
	wide		deep		wide		deep	
dataset	TD	BU	TD	BU	TD	BU	TD	BU
125k	27.84	38.95	7.3	9.25	27.57	39.05	7.24	9.20
250k	54.06	75.59	14.0	17.89	53.86	77.86	13.90	17.88
500k	103.57	144.4	27.16	34.86	106.25	159.04	27.01	35.16
1000k	191.54	266.79	52.67	67.78	210.57	330.68	53.11	70.67
2000k	335.64	469.19	-	-	388.25	615.69	-	-
4000k	585.42	850.36	-	-	596.48	981.58	-	-

	Zipf(0.7)				Zipf(0.9)			
	wide		deep		wide		deep	
dataset	TD	BU	TD	BU	TD	BU	TD	BU
125k	60.30	121.48	9.66	14.71	397.68	1.014.01	123.19	236.20
250k	165.2	342.78	22.46	36.08	987.44	1.945.08	432.32	392.52
500k	467.46	840.94	62.70	109.47	2.271.51	2.814.92	466.58	942.95
1000k	1.069.51	1.659.54	179.35	317.55	3.725.11	4.608.03	1.338.16	1.595.47
2000k	2.028.63	2.811.54	-	-	-	-	-	-
4000k	3.075.34	4.416.92	-	-	-	-	-	-

Table 5.3: The disk size of the inverted files (in kB), for the top-down (TD) and bottom-up (BU) algorithms. The deep data sets take less disk space than the wide data sets. Recall that deep trees (Figures 5.1-5.4) are constructed faster than wide trees.

Figure 5.1 shows the elapsed time of constructing the inverted files for uniform data sets and Figures 5.2, 5.3 and 5.4 for skewed data sets (with different skewness parameters). Note that a logarithmic scale is used on the vertical axis. The figures show that constructing deep trees is overall faster than constructing wide trees. This is a result of the recursion, on the node children, that is required in deep trees and as a consequence we decided to cap the number of these type of trees at $1000k$ because of the exponential growth of leaf values ($= l * c^{\text{maxdepth}}$). We also do the same for wide trees with skewness 0.9. The figures also show that an increase in the data sets does not imply an increase in construction time. Figure 5.1 shows a decrease in construction time between the 125k and 250k, and 1000k and the 2000k bottom-up data sets (wide). This decrease is also visible in Figure 5.3. The bottom-up data set (deep) shows a decrease in construction time between the 125k and 250k, and the 500k and 1000k. These dips are a side effect of the RandomTree function. In some cases the number of leaves and node children of a tree can have higher values in smaller data sets. The parameters l and c in the RandomTree function are upper bounds.

Example: Suppose that D_1 is tree in a $250k$ data set and D_2 is a tree in a $500k$ data set and let l_1 and c_1 be the leaf values and node children, respectively of D_1, and l_2 and c_2 of D_2. Then, we know that

$$l_1, l_2 \leq l \text{ and } c_1, c_2 \leq c, \text{ but it may be the case that } l_1 > l_2 \text{ and } c_1 > c_2.$$

As a result, smaller data sets can have an increased creation time. Table 5.3 shows the disk sizes of the inverted files for the synthetic data.

	min value	max value
data set cardinality Twitter	125,000	500,000
data set cardinality DBLP	125,000	4,000,000
leaf value cardinality Twitter	1	1,745,277
leaf value cardinality DBLP	1	267,142

Table 5.4: The statistics for real data. We use two different data sets: Twitter and DBLP. For Twitter, we do not have data sets higher than $500k$. The leaf value cardinalities depend on the string values in the data sets, e.g., no value in the Twitter data set has an integer representation higher than $1,745,277$.

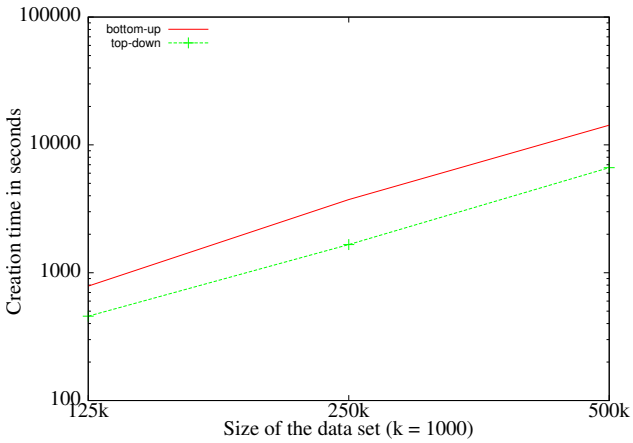


Figure 5.5: The creation time of the inverted file with the Twitter data set. The figure shows an exponential relationship between the different sizes. The top-down inverted file takes less time to construct.

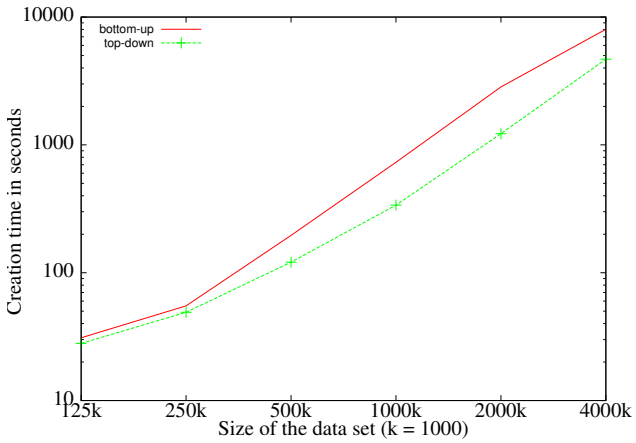


Figure 5.6: The creation time of the inverted file for DBLP data set. Again, we see an exponential relationship and that the top-down inverted file is created faster than the bottom-up.

5.3 Real data

Two different real data sets were used:

- tweets from Twitter; the tweets are in JSON format and contain information (e.g., messages, dates, user ids and urls) about 'Justin Bieber'. This data is also used in the Master's Thesis of J.Roijackers [Roi12], and
- the DBLP Computer Science Bibliography[5], as an XML database. This bibliography contains computer science related articles.

For a summary, see Table 5.4. The real data sets have strings as leaf values. An injective function was devised that converted the string based data sets to an integer based data sets, i.e., if two strings are equal, then their integer representation must also be equal.

5.3.1 Constructing the inverted file

For the real data, both algorithms also use different inverted files.

Figure 5.5 and 5.6 show the elapsed time of constructing the inverted files. In both cases, the inverted file for the bottom-up algorithm takes longer to construct. Also, constructing the inverted files for the

[5] http://www.informatik.uni-trier.de/~ley/db/

	Twitter		DBLP	
data set size	TD	BU	TD	BU
125k	1.265.472	2.344.588	49.580	191.304
250k	3.459.844	5.237.492	75.880	239.304
500k	6.856.564	8.917.884	267.992	488.636
1000k	-	-	476.020	1.050.528
2000k	-	-	764.784	1.339.644
4000k	-	-	1.893.560	2.572.796

Table 5.5: The disk size of the inverted files (in kB). The Twitter data sets takes significantly more disk space than the DBLP data set. This can be explained by the leaf count of the data sets. Table 5.4 shows that the Twitter sets contains approximately 6.5x more leaves than the DBLP data set.

Twitter data set with $500k$ trees takes almost the same amount of time as constructing the inverted file for the DBLP data set with $4000k$ trees. The cause is the size of the inverted lists retrieved from the inverted file. Long lists take more time to update than small lists. The Twitter data set shows a long tail distribution (e.g., frequency of popular tweets, number of posts per user). This results in long inverted lists. The DBLP data sets shows a more uniform distribution (the lists are relatively small). Table 5.5 shows the disk sizes of the inverted files for real data.

5.4 Generating queries

For queries, we took random trees from our data sets and stored them in main memory. The number of queries is set to 100, from which 50 are contained in the data sets (positive) and 50 are distorted (negative). Distorting a tree here means adding leaf values at an arbitrary level of that tree.

5.4.1 Unit of measurement

The unit of performance measurement in our experiments is the elapsed time of sequentially executing those queries ten times. From those ten, we exclude the minimum and maximum times. Also, we noticed that the maximum, in most of the cases, was the first run out of the ten runs (due to overhead setting up the Java Virtual Machine).

5.4.2 Bloom filter

We conduct an experiment with the Bloom filter, using the top-down algorithm, for synthetic data. The additional cost (in disk space) is the space storage for the Bloom filter. Every query and tree in the inverted file has a Bloom filter in the form of a boolean array with length 20. An important aspect of the Bloom filter is that the number of bits set to 1 should be relatively small. If the number of bits set to 1 is high, it is more likely that less records will be pruned. Figure 5.7 shows the impact of adding values in the Bloom filter (See Appendix B for source code and hash functions).

Queries

For queries with a Bloom filter, we evaluate deep trees since preliminary analysis showed that wide trees are not well-suited for Bloom filters. The data set contains $100k$ trees and the average depth is

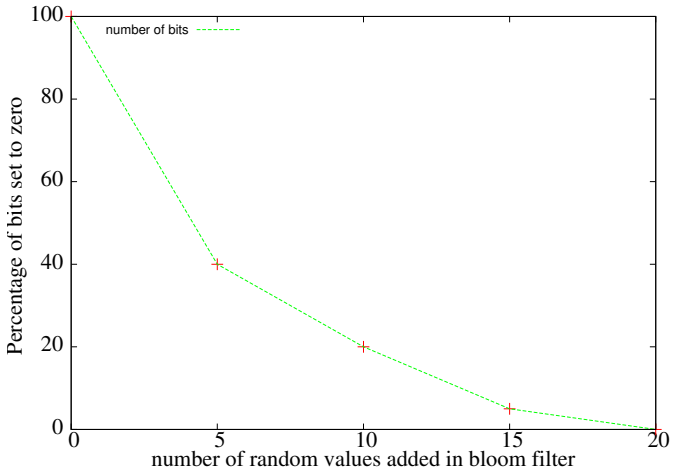


Figure 5.7: An example of a Bloom filter that contains values taken from the range $[0, ..., 100000]$. The figure shows that more than 40% of the bits remains 0 if the number of elements added is less than 5. The figure also shows that every bit is set to 1, if 20 elements are added in the Bloom filter.

set to 7. The queries are a subset of the data set but with a little twist. Every query is distorted, i.e., random leaves are added at the lowest level of the query. We want to investigate if the Bloom filter spots false matches relatively fast without traversing the entire tree.

5.5 High level overview of preprocessing

Figure 5.8 shows procedure prior to querying. The process starts by extracting all leaf values in the trees, of the data set and storing them in a hash table. The keys are the leaves and the values are the number of occurrences in the data set. The second step is updating the data set with Bloom filters (merely for the top-down algorithm). Every tree in the set gets a Bloom filter which contains the least frequent leaf values of the tree. The last step is building the inverted file. We refer the reader to Appendix A, for more implementation details.

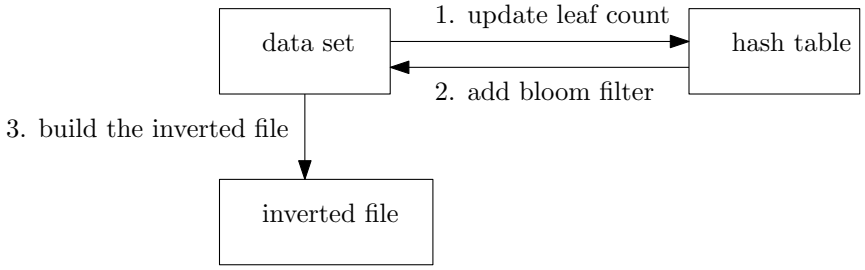


Figure 5.8: A high level overview of the construction regarding the most frequent leaves and the inverted file. Note that the steps 1-3 are not executed every time, but merely after added new trees in the data set.

5.6 Summary

In this chapter we explained the system parameters and set-up used for the experiments. The synthetic data (uniform and Zipfian) consist of wide and deep trees. The results, of measuring construction time of the inverted files, show that wide trees are constructed faster than deep trees.

The real data is taken from Twitter and the DBLP Computer Science Bibliography. For both data sets, constructing the inverted file for the top-down algorithm is faster than the bottom-up algorithm. For queries, we took random trees from our data sets and stored them in main memory. Also, we added Bloom filters in the queries. We concluded the chapter with a high level overview on how to construct the inverted file.

6 Empirical analysis

In this chapter we present our empirical results. Section 6.1 discusses the synthetic data. Recall we use uniform and skewed distribution of the leaf values in all data sets. Then, in Section 6.2, we present the results of the realistic data sets, i.e., Twitter tweets and the DBLP Computer Science Bibliography. In Section 6.3, we compare positive and negative queries. Recall that positive queries are subsets and negative queries return zero results. In Section 6.4 we present an experiment with the Bloom filter. We conclude the chapter with the discussion of the results.

6.1 Experiment 1: Synthetic data

In this Section we present the results of the synthetic data experiments. The synthetic data consists of uniform and skewed (i.e., Zipfian distributed) data sets.

6.1.1 Uniform

Figure 6.1 and 6.2 show the results of the uniform data. The figures show that an increase in size does not always imply an increase in querying time. Also, caching shows no effect on the uniform data sets.

6.1.2 Skewed

For the skewed data set we use the parameters 0.5 (Figures 6.3 and 6.4), 0.7 (Figures 6.5 and 6.6) and 0.9 (Figures 6.7 and 6.8). Note that a logarithmic scale is used on the vertical axis, for Figures 6.5-6.8. A direct observation is that as the skewness parameter increases, the querying time also increases. Figures 6.7 and 6.8 show that the length of the inverted lists grows exponentially fast with an increasing value of the skewness parameter. The wide data sets (Figures 6.3, 6.5 and 6.7) do not show a decrease in querying time with caching. Also, there is barely any difference between the top-down and bottom-up algorithm. Figure 6.5 shows there is a slight decrease in querying time with caching. For the deep data sets, with skewness 0.7 (Figure 6.5), caching shows a slight decrease in querying time. Also, Figure 6.8 shows that the top-down algorithm (with and without caching) performs better than the bottom-up algorithm.

A general observation is that Figures 6.1-6.6 show noise/artifacts, i.e., dips in querying time. The dips only occur if the querying time is less than 1 second. An explanation for this phenomenon lies in the

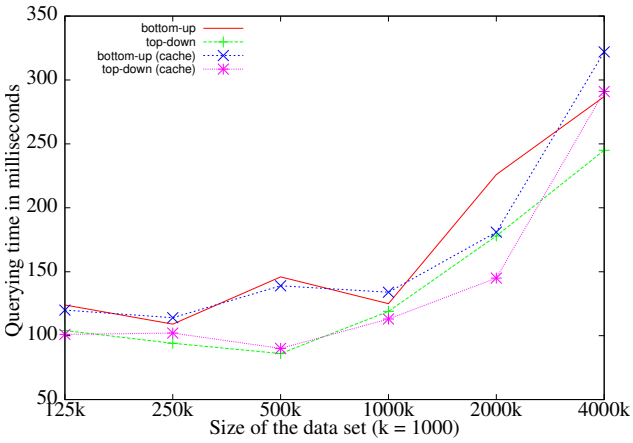


Figure 6.1: The cumulative, average querying time on uniform, wide data sets (Experiment 1). The figure shows that the bottom-up algorithm performs worse than the top-down algorithm. Also, there is not a significant increase in querying time with respect to the data sets.

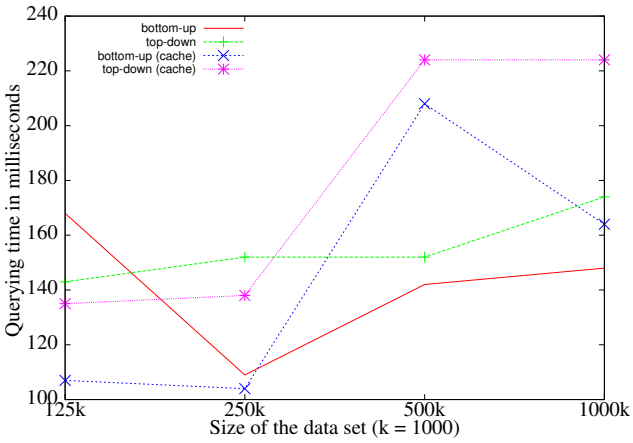


Figure 6.2: The cumulative, average querying time on uniform, deep data sets (Experiment 1). The figure shows noise with the bottom-up algorithm.

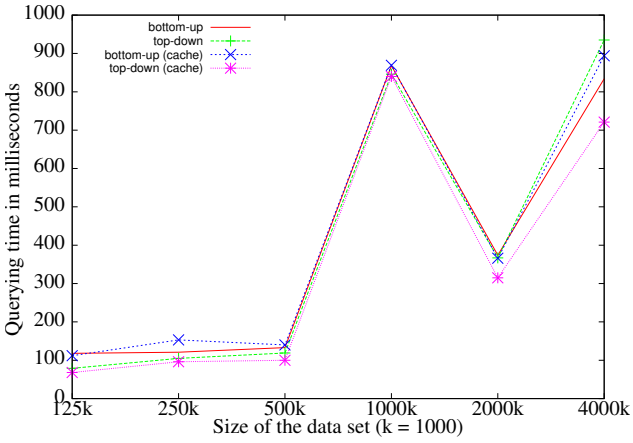


Figure 6.3: The cumulative, average querying time on wide data sets with skewness 0.5 (Experiment 1). The figure shows that both algorithms have the same overall querying time. Also, there is a decrease in querying time from the 1000k to the 2000k data set, due to noise.

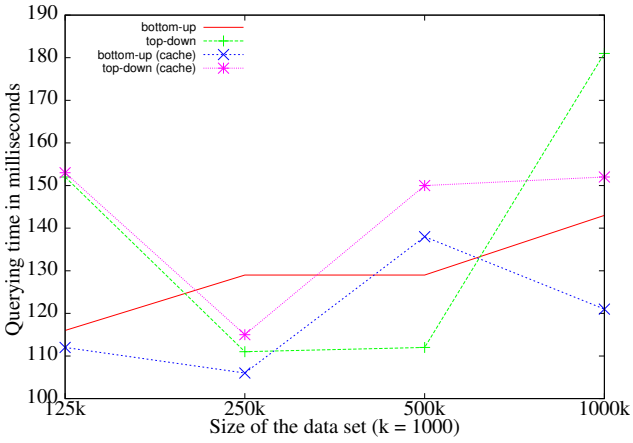


Figure 6.4: The cumulative, average querying time on deep data sets with skewness 0.5 (Experiment 1). Again, we see the same noise (skewness 0.5) which makes it hard to compare both algorithms.

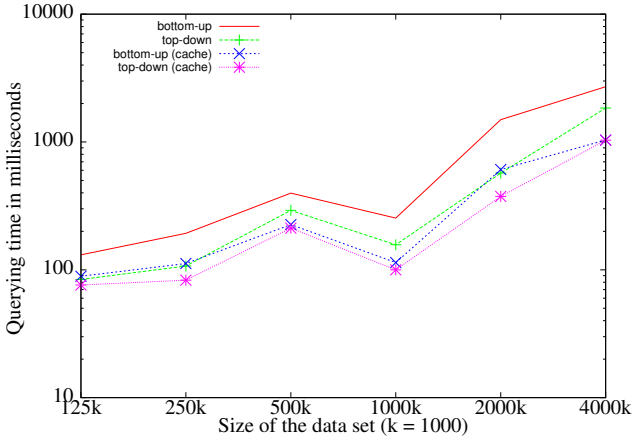


Figure 6.5: The cumulative, average querying time on wide data sets with skewness 0.7 (Experiment 1). The figure shows a slight decrease in querying time with caching. Also, both data sets show a decrease in querying time from the 500k to the 1000k data sets.

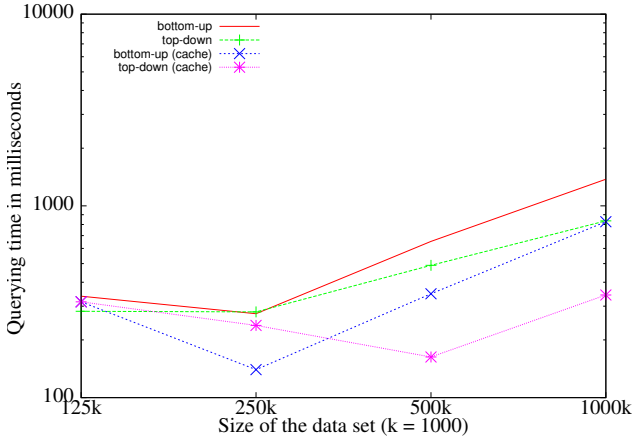


Figure 6.6: The cumulative, average querying time on deep data sets with skewness 0.7 (Experiment 1). The figure shows a slight decrease in querying time with caching. The figure also shows noise with caching, for both algorithms. In this case the noise occurs on different data sets.

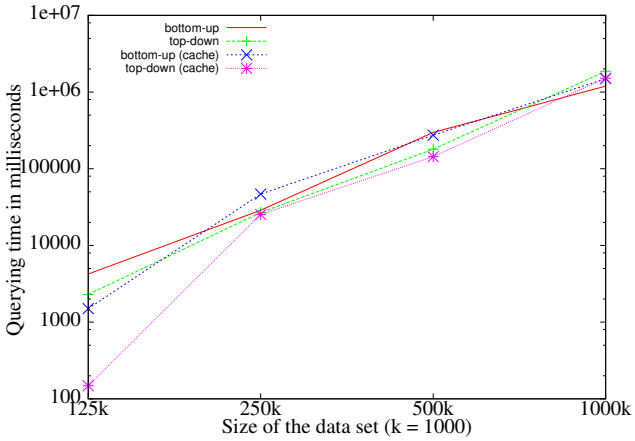


Figure 6.7: The cumulative, average querying time on wide data sets with skewness 0.9 (Experiment 1). The figure shows a significant increase in querying time with respect to the sizes. The top-down algorithm, with caching, shows promising results with the smallest data set, however this is not the case for the other sizes.

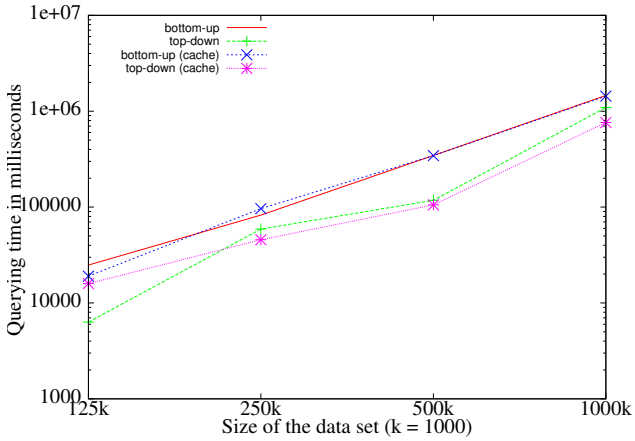


Figure 6.8: The cumulative, average querying time on deep data sets with skewness 0.9 (Experiment 1). The figure shows that the top-down algorithm is overall faster than the bottom-up algorithm. The figure also shows that caching does not make any difference in most of the cases. With caching, there is a slight decrease in querying time with sizes greater than 250k (with top-down algorithm).

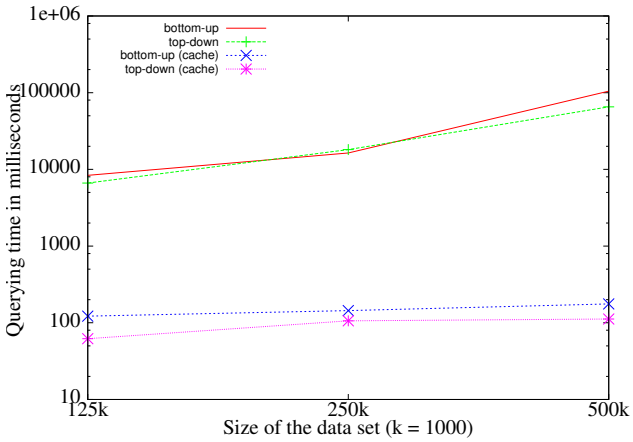


Figure 6.9: The cumulative, average querying time on the Twitter data set (Experiment 2). The figure shows a significant decrease in querying time with caching enabled for both algorithms. Also, the top-down algorithm with caching has the least querying time.

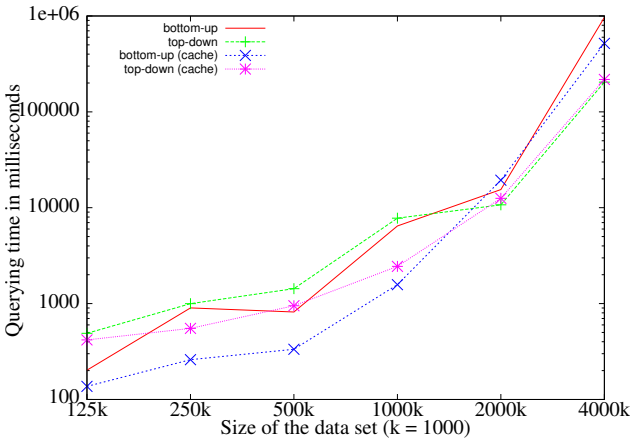


Figure 6.10: The cumulative, average querying time on the DBLP data set (Experiment 2). Initially, the bottom-up algorithm with caching enabled has the least querying time, but this drop when the size exceeds the 2000k.

number of leaf values and node children of a tree (in a data set). In Section 5.2, we explained the number of leaves and node children of a tree depend on a random function. Smaller data sets can have higher leaf values and more node children than larger data sets. As a result, the querying time of smaller data sets is in some cases higher than larger data sets. We do not see this phenomenon with highly skewed data sets (Figures 6.7-6.9). Uniformly distributed data sets have relatively small inverted lists and the most expensive part of the querying time is retrieving the inverted lists from the inverted file. Highly skewed data have longer inverted lists and the intersection part of the querying time dominates the retrieval time of the inverted lists.

6.2 Experiment 2: Real data

In this Section we present our real data results. Note that a logarithmic scale is used on the vertical axis. Figure 6.9 shows the results of the Twitter data set. The figure shows an increase by a factor of 100 with caching. In most cases, the top-down algorithm performs better than the bottom-up. Figure 6.10 shows the results of the DBLP data set. The bottom-up algorithm with caching shows the lowest querying time with data sets $\leq 1000k$. Also, both algorithms show a significant increase in querying time for larger data sets.

6.3 Experiment 3: Positive vs negative queries

Table 6.1 shows the cumulative, average querying time of the positive and negative queries (discussed in Section 5.4), for the data sets of size $500k$. Recall that positive queries are subsets and negative queries return zero results. Our first observation is that $\frac{15}{24}$ experiments, the negative queries are executed faster than the positive queries. The bottom-up algorithm performs much worse when evaluating negative queries. Also, the top-down algorithm spots negative queries faster. The second observation is that $\frac{9}{12}$ data sets caching performs better. This is shown in the speed-up, which is defined as $\frac{\text{no caching}}{\text{caching}}$ (especially with the Twitter data sets caching shows a significant improvement).

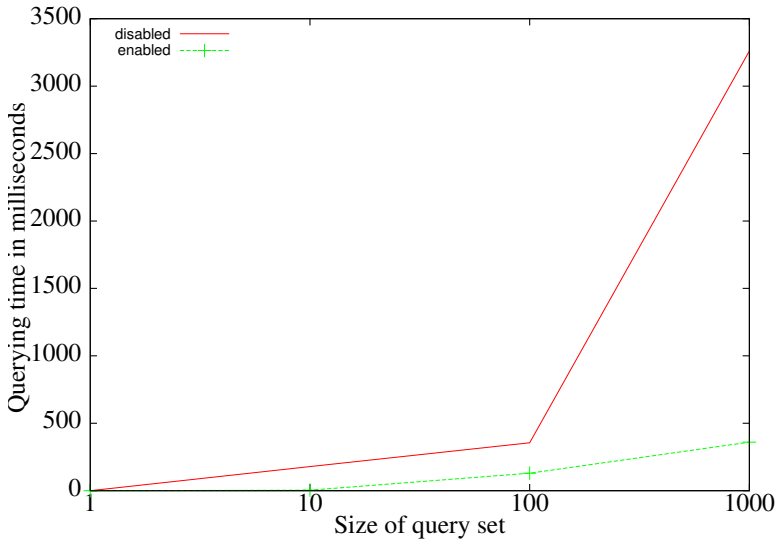


Figure 6.11: The cumulative, average querying time with the top-down algorithm (Experiment 4). The data set contains 100,000 trees. Every tree is distorted and the average depth is 7. The figure shows that the querying time increases significantly with 1000 queries, as apposed to the Bloom filter, there we see a linear increase with respect to the query size.

The third observation is the ratio $\alpha = \frac{\text{TD}}{\text{BU}}$. The top-down algorithm perform in $\frac{13}{24}$ experiments better than the bottom-up algorithm. This is not significant, but what is interesting is that these are all negative queries in which the top-down algorithm performs better.

6.4 Experiment 4: The impact of Bloom filter

We implemented the Bloom filter with both synthetic and real data. Analyses shows that the Bloom filter performs better with data sets where every tree is distorted. Figure 6.11 shows that the Bloom filter spots false matches faster than without the Bloom filter.

6.5 Discussion

In the previous sections we presented the results of the experiments. We showed that the cumulative, average querying of Experiment 1 was relatively fast, for the uniform data sets ($\leq$ 350 ms). The results of the uniform data sets in Experiment 1 showed artifact, i.e., larger data sets have lower querying time than smaller data sets. We explained that smaller data sets can have higher leaf values and more node children than larger data sets. As a result, the querying time of smaller data sets was in some cases higher than larger data sets. We also explained that his only occurs with uniformly distributed data. Also, caching a subset of the payloads has little effect on the uniform data sets. The leaf values are uniformly distributed, therefore it is unlikely that a cached leaf value is fetched frequently. As a result caching merely caused overhead instead of gain in querying time. For the skewed data set in Experiment 1, we saw that an increase in the skewness parameter resulted in an increase in querying time. The data sets with skewness 0.9 in Experiment 1 showed that the length of the inverted file grew exponentially fast with an increasing skewness value. The intersection of relatively large inverted lists takes longer and as a result the querying time also increases. Both, the uniform and skewed data in Experiment 1 showed noise when the querying time was relatively low. The data sets in Experiment 1,

		uniform						Zipf(0.5)						Real					
		wide			deep			wide			deep			Twitter			DBLP		
		TD	BU	α	TD	BU	α	TD	BU	α	TD	BU	α	TD	BU	α	TD	BU	α
no caching	+	1.18	1.02	1.2	1.39	0.83	1.7	1.9	1.8	1.1	5.58	4.04	1.4	64.1	74.11	0.9	4.42	2.51	1.8
	-	0.4	1.15	0.4	0.94	1.53	0.6	0.1	4.18	0	1.18	7.71	0.2	9.76	14.64	0.7	0.62	0.79	0.8
cache	+	1.46	0.75	2	1.76	0.92	1.9	0.53	0.84	0.6	2.64	2.28	1.2	0.79	0.69	1.1	1.94	0.98	2
	-	0.56	0.83	0.7	1.15	1.69	0.7	0.07	3.09	0	0.67	4.96	0.1	0.1	0.88	0.1	0.4	0.36	1.1
speed-up	+	0.8	1.36	0.6	0.78	0.9	0.9	3.6	2.14	1.7	2.11	1.78	1.2	81.1	107.4	0.8	2.28	2.56	0.9
	-	0.71	1.39	0.5	0.82	0.91	0.9	1.4	1.35	1	1.76	1.55	1.1	97.6	16.64	5.9	1.55	2.19	0.7

Table 6.1: The cumulative, average querying time on negative and positive queries in ms (for 500k data sets, Experiment 3). The table shows a comparison between the top-down and bottom-up algorithm, and caching enabled/disabled. The speed-up is the ratio of no caching to caching. The α-value is the ration of top-down to bottom-up. The Twitter data set shows the most promising result regarding the speed-up. For the α-value, we see no significant difference, in querying time, between the top-down and the bottom-up algorithm.

with skewness 0.9, do not show any noise. This is quite logical since the order of magnitude is almost a factor 10 with respect to the size of the data set.

The Twitter data set in Experiment 2 showed promising results with caching. There, we saw an speed-up by a factor 100. The most frequent leaves in the data set were tweets that were post during the weekends and a subset of users that posted on a regular basis. Caching these items increased the performance of the inverted file. Also, the top-down algorithm performs better than the bottom-up in most cases. We elaborate this result in the following paragraph (Experiment 3). The results of the DBLP data sets in Experiment 2 showed that caching had little effect. An explanation of this is the distribution on the leaf values which tend towards uniform, e.g., the gaps between the most frequent leaves became smaller with larger data sets. Also, both algorithms showed a significant increase in querying time for larger data sets.

In Experiment 3, we focussed on the data sets of size $500k$. There, we did a comparison between positive and negative queries. Negative queries are overall executed faster than positive queries. This is expected, since positive queries imply that the complete trees have been visited, as opposed to the negative queries. There, we see that if no match is found at the current level of a tree, the evaluation immediately stops. The comparison also showed that the bottom-up algorithm performs much worse when evaluating negative queries. A logical explanation is the distorting function of a tree. The distorting starts at the top of a tree and is bounded by a stopping probability (the same as the one described in Section 5.2). As a result, it may be the case that the upper part of a tree is distorted and the lower part of a tree remains the same. The top-down algorithm spots negative queries faster because the top levels are distorted. This also explains all the cases in which the top-down algorithm outperforms the bottom-up algorithm. Finally, we implemented a Bloom filter with both synthetic and real data, in Experiment 4. We showed that false matches are spotted faster with the Bloom filter.

7 Conclusion

In this thesis we studied the problem of containment queries on nested sets of atomic objects. We showed that nested sets can also be represented as trees and with that representation in mind, we introduced two novel algorithms for evaluating containment queries on nested sets. The algorithms use an inverted file as an approach for evaluating subset containment. The first algorithm uses a top-down approach that starts by evaluating a query at the root node and continues with its children nodes. The second algorithm uses a bottom-up approach, i.e., starts exploring a query at the leaf nodes and propagates up to the root node. Analytic and empirical analyses shows that both algorithms have the same general runtime behaviour. Generally, they both showed fast response times with querying of synthetic and realistic data. We also saw a significant decrease in querying time when we cached a subset of the payloads, on the skewed data sets. In general, both solutions have difficulties handling highly skewed data.

Future work

In Sections 4.5 and 4.6, we discussed several extensions to both algorithms. A future work might involve implementing those extensions and doing the same empirical analysis as in Chapter 6. Also, the empirical analyses can be extended with other real data. In Section 4.4 we discussed the notion of caching with respect to the data set. Another natural notion of caching, that can be implemented, is with respect to an observed query workload. As opposed to caching, in [TPVS06] and [TBV+11], the authors present additional data structures for the inverted files for further pruning the number of disk pages that need to be retrieved from the hard disk. It is worth investigating if additional data structures can support our solutions, particularly for skewed data sets. Also, to reduce storage and retrieval costs, compression techniques for the payloads used in both solutions should be investigated. We also discussed the notion of bloom filters and how we can use them to assist us in evaluating containment queries, i.e., prune false matches without investigating the complete tree structure. In [HAB12] and [TP10] they present different approaches for measuring structural similarities of semistructured data and hashing tree-structured data. These approaches are also worth investing, for pruning purposes in our context.

Appendix: Description of the source code

In this Appendix we give a high level description of the source code. We elaborate important packages, classes and discuss the programs we used in our empirical analyses (Chapter 5).

A.1 Packages

The Java project consists of the following packages:

- main. This package contains the implementation of both algorithms.
- tokyocabinet. This package contains classes that use the embedded database Tokyo Cabinet.
- util. A utility package for generating data, statistics and other static methods that can be accessed without instantiating classes.
- xml. This package contains classes for converting an XML database to our schema.
- twitter. This package contains classes for converting Twitter (i.e., JSON) data to our schema.

A.2 Classes

In this section we highlight a few relevant classes:

- Tree. This class represents a tree. It contains attributes and methods such as :
  - Integer *id*: returns the identifier of the Record class.
  - Set<Integer> $leafValues$: a set with the leaf values of the Record class.
  - Set<Record> $childNodes$: a set with the children of the Record class.
  - Boolean $isRoot$: returns true if the Record is at the root node.
  - Integer $computeMaxDepth$: computes the maximum depth of the Record.
- PairList. This class is the payload of a retrieved leaf value. It consists of an *id*, i.e., the leaf value and a List<PairDisk>. In Figure 3.2 we give an example of an inverted file. In that example a PairList of the leaf value 1 is:

(201,<202,203>), (203,<204,208>), (204,<>)

- PairDisk. A PairDisk is a part of the payload in a PairList, e.g., (201,<202,203>), (203,<204,208>) and (204,<>) are three PairDisks of the PairList above.
- TDInvertedFileDisk. This class implements the top-up algorithm presented in Section 4.3.1.
- BUInvertedFileDisk. This class implements the bottom-up algorithm presented in Section 4.3.2.

A.3 Programs

These are the 8 programs used for this thesis, divided into different groups:

- Database: These classes create the records up to $4,000,000$.

  A_1 JSONRecConverter: this class converts Twitter tweets into the Record format used by the algorithms.

  A_2 XMLRecConverter: this class converts a XML database into the Record format used by the algorithms.

  A_3 TCSaveSyntDataProgram: (TC is an abbreviation of Tokyo Cabinet): this class creates synthetic data with an uniform or Zipfian distribution on the leaf values. Also, parameters such as the maximum leave count or maximum child nodes can be provided.

- Additions of the databases: These classes add extra information to the databases. Note that these classes are only created once.

  B_1 TCSaveMostFreqDataProgram: this class builds a hash table that contains the most frequent leaves. It requires the database and the number of leaves. E.g., if the number = 100, then the 100 most frequent leaves will be stored in a hash table with key: the leaf and value: the number of times the key occurs in the database.

  B_2 TCUpdateBloomFilterProgram: this class add a bloom filter to every record in the database. Bloom filters are only used in some cases, therefore we use a separate class to avoid the overhead in the other cases.

- Constructing the inverted files.

  C_1 TCConstructIFProgram: This class constructs the inverted file from the provided databases. Three types of inverted files (we distinguish for fair comparisons and overhead reasons) can be constructed: a top-down inverted file, a bottom-up inverted file and a top-down inverted file with a bloom filter.

- Querying: These classes are used for querying.

  D_1 TDViewSyntDataIFProgram: this class evaluates the queries with the top-down algorithm. It requires a database (to fetch the queries from) and a top-down inverted file or a top-down inverted file with a bloom filter.

  D_2 BUViewSyntDataIFProgram: this class evaluates the queries the bottom-up algorithm. It requires a database (to fetch the queries from) and a bottom-up inverted file.

All classes are console based and have their own parameters. E.g., create 1000 records with relative deep queries, maximum leaf count at each level is 3, the maximum child nodes is 2 and the distribution is uniform. The result is shown in the following command:

java -cp proj.jar tokyocabinet.TCSaveSyntDataProgram -h data.out 1000 3 2 0.2 false

where:

- java -cp is the JVM command,
- proj.jar is the executable jar file that contains the entire project,
- tokyocabinet.TCSaveSyntDataProgram is the name of the package and class,
- -h is a prefix, no execution takes place without his parameter,
- data.out is the newly created database with 1000 records,
- 0.2 is the deepness of the record (this parameter was discussed in section 5.2), and
- false implies uniform distribution.

The other parameters, i.e., 1000, 3 and 2 are respectively the number of records, maximum leaf count and maximum child nodes.

The programs cannot be executed independently. There are 9 executions possible (in total 12 but we exclude the combinations with the bloom filter and bottom-up algorithm since it is only implemented in the top-down algorithm):

$$[A_1 \text{ or } A_2 \text{ or } A_3] \cdot [B_1 \text{ or } B_2] \cdot C_1 \cdot D_1$$

$$[A_1 \text{ or } A_2 \text{ or } A_3] \cdot B_1 \cdot C_1 \cdot D_2$$

e.g., the following sequence generates a synthetic data set with bloom filters, builds the inverted file and uses the top-down algorithm for evaluation:

$$A_3 \cdot B_2 \cdot C_1 \cdot D_1$$

Appendix: Bloom filter implementation

In this Appendix we present the source code of the bloom filter, that is implemented in the top-down algorithm, in Java.

Listing B.1: Source code bloom filter (in Java)

```
/**
  Bloom filter: hash functions h1,...,h4 with formula (from Cormen et al.):
    - h(a,b,k) = ((a*k+b) % p) % M, where:
    - p is large prime (i.e. larger than any key k),
    - a in [0,p-1],
    - b in [1,p-1], and
    - M is bucket size.
 */
public class BloomFilter{
        private static final int M =  20;
        private static final int p = 15485863;

        private boolean[] hashTable;
        private int size = 0;


        public BloomFilter(){
                hashTable = new boolean[M];
        }

        public void add(int val){
                if(!contains(val)){
                        hashTable[h1(val)] = true;
                        hashTable[h2(val)] = true;
                        hashTable[h3(val)] = true;
                        hashTable[h4(val)] = true;
                        size++;
                }
        }
```

```
        public boolean[] getHashTable(){
                return hashTable;
        }

        public int size(){
                return size;
        }

        public boolean contains(int val){
                return hashTable[h1(val)]
                && hashTable[h2(val)]
                && hashTable[h3(val)]
                && hashTable[h4(val)];
        }

        private static int h1(int val){
                return ((val*307 + 67) %p)%M;
        }

        private static int h2(int val){
                return ((val*173 + 9739) %p)%M;
        }

        private static int h3(int val){
                return ((val*563 + 19) %p)%M;
        }

        private static int h4(int val){
                return ((val*37 + 31) %p)%M;
        }

}
```

Bibliography

[BM02] Andrei Broder and Michael Mitzenmacher. Network Applications of Bloom Filters: A Survey. In *Internet Mathematics*, volume 1, pages 636–646, 2002.

[Boe11] Melle Boersma. *A study of nested-relational joins in mediator-based distributed environments (M.Sc. thesis)*. Eindhoven University of Technology, Department of Mathematics and Computer Science, 2011.

[CD99] James Clark and Steve DeRose. XML Path Language (XPath) version 1.0. Recommendation, World Wide Web Consortium, November 1999. See http://www.w3.org/TR/xpath.html.

[CM10] J. Shane Culpepper and Alistair Moffat. Efficient set intersection for inverted indexing. *ACM Trans. Inf. Syst.*, 29:1–25, December 2010.

[CSRL01] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[DKS08] Roman Dementiev, Lutz Kettner, and Peter Sanders. Stxxl: standard template library for xxl data sets. *Softw. Pract. Exper.*, 38(6):589–637, 2008.

[GC07] Gang Gou and Rada Chirkova. Efficiently querying large XML data repositories: A survey. *IEEE Trans. on Knowl. and Data Eng.*, 19(10):1381–1403, October 2007.

[GJ00] Georgia Garani and Roger Johnson. Joining nested relations and subrelations. *Inf. Syst.*, 25(4):287–307, June 2000.

[GKM09] Michaela Götz, Christoph Koch, and Wim Martens. Efficient algorithms for descendant-only tree pattern queries. *Inf. Syst.*, 34(7):602–623, November 2009.

[HAB12] Sven Helmer, Nikolaus Augsten, and Michael Böhlen. Measuring structural similarity of semistructured data based on information-theoretic approaches. *The VLDB Journal*, pages 1–26, to appear, 2012.

[HD11] Marouane Hachicha and Jerome Darmont. A survey of XML tree patterns. *IEEE Transactions on Knowledge and Data Engineering*, to appear, 2011.

[HM97] Sven Helmer and Guido Moerkotte. Evaluation of main memory join algorithms for joins with subset join predicates. In *Proc. of the 23rd Conf. on Very Large Data Bases (VLDB)*, pages 386–395. Morgan Kaufmann Publishers Inc., 1997.

[HM03] Sven Helmer and Guido Moerkotte. A performance study of four index structures for set-valued attributes of low cardinality. *The VLDB Journal*, 12(3):244–261, October 2003.

[KAR12] Yusaku Kaneta, Hiroki Arimura, and Rajeev Raman. Faster bit-parallel algorithms for unordered pseudo-tree matching and tree homeomorphism. *Journal of Discrete Algorithms*, 14:119–135, 2012.

[Kil92] Pekka Kilpeläinen. *Tree Matching Problems with Applications to Structured Text Databases.* PhD thesis, Department of Computer Science, University of Helsinki, 1992.

[KP04] Georgia Koloniari and Evaggelia Pitoura. Filters for XML-based service discovery in pervasive computing. *The Computer Journal*, 47(4):461–474, 2004.

[Mam03] Nikos Mamoulis. Efficient processing of joins on set-valued attributes. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 157–168. ACM, 2003.

[MGM02] Sergey Melnik and Hector Garcia-Molina. Divide-and-conquer algorithm for computing set containment joins. In *Proceedings of the 8th International Conference on Extending Database Technology: Advances in Database Technology*, pages 427–444. Springer-Verlag, 2002.

[MGM03] Sergey Melnik and Hector Garcia-Molina. Adaptive algorithms for set containment joins. *ACM Trans. Database Syst.*, 28:56–99, March 2003.

[Roi12] John Roijackers. *Bridging SQL and NoSQL (M.Sc. thesis).* Eindhoven University of Technology, Department of Mathematics and Computer Science, 2012.

[RPNK00] Karthikeyan Ramasamy, Jignesh M. Patel, Jeffrey F. Naughton, and Raghav Kaushik. Set containment joins: The good, the bad and the ugly. In *Proceedings of the 26th International Conference on Very Large Data Bases*, pages 351–362. Morgan Kaufmann Publishers Inc., 2000.

[SCF+07] Jérôme Siméon, Don Chamberlin, Daniela Florescu, Scott Boag, Mary F. Fernández, and Jonathan Robie. XQuery 1.0: An XML query language. W3C recommendation, W3C, January 2007. http://www.w3.org/TR/2007/REC-xquery-20070123/.

[ST99] Ron Shamir and Dekel Tsur. Faster subtree isomorphism. *Journal of Algorithms*, 33:267–280, 1999.

[TBV+11] Manolis Terrovitis, Panagiotis Bouros, Panos Vassiliadis, Timos Sellis, and Nikos Mamoulis. Efficient answering of set containment queries for skewed item distributions. In *Proceedings of the 14th International Conference on Extending Database Technology*, EDBT'11, pages 225–236. ACM, 2011.

[TP10] Shirish Tatikonda and Srinivasan Parthasarathy. Hashing tree-structured data: Methods and applications. In *Proceedings of the International Conference on Data Engineering (ICDE'10)*, pages 429–440. IEEE, 2010.

[TPVS06] Manolis Terrovitis, Spyros Passas, Panos Vassiliadis, and Timos Sellis. A combination of trie-trees and inverted files for the indexing of set-valued attributes. In *Proceedings of the 15th ACM international conference on Information and knowledge management*, CIKM '06, pages 728–737. ACM, 2006.